

David Lawrence and Mark England

David Lawrence and Mark England



The Amstrad Companion

David Lawrence and Mark England

First published 1986 by:
Sunshine Books (an imprint of Scot Books Ltd.)
12-13 Little Newport Street
London WC2H 7PP

Copyright © David Lawrence and Mark England

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording and/or otherwise, without the prior permission of the Publishers.

Sunshine Books, makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchant ability or fitness for any particular purpose. Further, Sunshine Books reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Sunshine Books to notify any person of such revision or changes.

British Library Cataloguing in Publication Data

Lawrence, David, 1949-

The 8256 companion.

1. Amstrad 8256 (Computer)

I. Title II. England, Mark

004.165 QA76.8.A/

ISBN 0-946408-95-5

Cover photograph by Angus Thompson.
Typeset and Printed in England by the
Sovereign Printing Group.

Contents in detail

	Introduction	1
1	CP/M	3
1.1.	The operating system	3
1.2.	What is CP/M	6
1.2.1.	CP/M Plus	6
1.3.	Some CP/M concepts	6
1.3.1.	Physical and logical disc drives	7
1.3.2.	Physical and logical devices	7
1.3.3.	User groups	9
1.3.4.	The ramdisc	11
1.4.	Structuring a disc for CP/M	12
1.4.1.	Creating a start-up disc	12
1.4.1.1.	<i>Creating a PROFILE.SUB file</i>	14
1.4.1.2.	<i>Customising PROFILE.SUB</i>	16
1.5.	Working discs	17
1.6.	Protecting files	19
1.7.	Single purpose discs	20
1.8.	A table of common CP/M usage	21
1.9.	Table of recommended CP/M file extensions	24
1.10.	CP/M editing keys	24
2.	Basic	27
2.1.	BASIC and the PCW	28
2.2.	Entering the BASIC programs	29
2.3.	Testing programs	29
2.4.	BANKER	31
2.4.1.	Saving the program	32
2.4.2.	Initialisation	33
2.4.3.	The Program Menu	35
2.4.4.	Entering new items	36
2.4.5.	Displaying the statement	39
2.4.6.	Data Files	42
2.4.7.	Examining and deleting items	43
2.5.	ACCOUNTANT	45
2.5.1.	Initialisation	46
2.5.2.	The main menu	46
2.5.3.	Credit or Debit?	48
2.5.4.	The type of item	48
2.5.5.	Entry of single items and main headings	49
2.5.6.	Entering a sub-heading	51
2.5.7.	Data files	52
2.5.8.	Changes to items	53
2.5.9.	Deleting items	54

	2.5.10.	Displaying the accounts	55
2.6.		UNIFILE	58
	2.6.1.	Setting up the structure of the file	58
	2.6.2.	Starting the program	60
	2.6.3.	Menu	61
	2.6.4.	Stopping the program	62
	2.6.5.	Saving data	62
	2.6.6.	Loading data	63
	2.6.7.	A better way of searching	64
	2.6.8.	Inserting an item	66
	2.6.9.	Making entries to the file	66
	2.6.10.	Identify items in a single entry	68
	2.6.11.	Searching for items in the file	69
	2.6.12.	Deleting entries	72
	2.6.13.	Changing entries	72
2.7.		NNUMBER	74
	2.7.1.	Initialisation	75
	2.7.2.	Menu	76
	2.7.3.	Data files and Error Handling	77
	2.7.4.	Binary search	78
	2.7.5.	Inserting items into the main dictionary	79
	2.7.6.	Entering items for the dictionary	79
	2.7.7.	The Search routine	80
	2.7.8.	Deleting an item	80
	2.7.9.	Copying items into the "current" list	82
	2.7.10.	Displaying the current list	83
	2.7.11.	Search and delete from the current list	84
	2.7.12.	Initialising the current list	85
2.8.		CARDINDX	85
	2.8.1.	Initialisation	87
	2.8.2.	Print title	88
	2.8.3.	Opening menu	88
	2.8.4.	Main menu	89
	2.8.5.	Error trapping	90
	2.8.6.	Ask user to confirm input	91
	2.8.7.	Get file name from user	91
	2.8.8.	Create new file	92
	2.8.9.	All field statements for l= 1 to 8	94
	2.8.10.	Open existing file	95
	2.8.11.	Add new record to file	96
	2.8.12.	Input record from keyboard	97
	2.8.13.	Write record to file	99
	2.8.14.	Search file for record	102

	2.8.15.	Find first record matching key	106
	2.8.16.	Next record matching key	106
	2.8.17.	Previous item matching key	109
	2.8.18.	Last item matching key	110
	2.8.19.	Delete record	111
	2.8.20.	Print card index	112
2.9.	BUDGET		113
	2.9.1.	Data files	114
	2.9.2.	Menu	115
	2.9.3.	Initialisation	117
	2.9.4.	Income	119
	2.9.5.	Input of payments	120
	2.9.6.	Update budget	121
	2.9.7.	Display Figures	123
	2.9.8.	Find budget head	128
	2.9.9.	Changes	129
	2.9.10.	Delete budget head	131
	2.9.11.	Update month	132
	2.9.12.	Set up what-if arrays	134
	2.9.13.	Testing the program	135
2.10.	TRIVIA		136
	2.10.1.	Setup system	137
	2.10.2.	Error	139
	2.10.3.	New types	139
	2.10.4.	Menu	141
	2.10.5.	Entry of new items	142
	2.10.6.	Binary search	144
	2.10.7.	Insert item	145
	2.10.8.	Data files	145
	2.10.9.	User search/delete	146
	2.10.10	Questions	148
	2.10.11	Score	151
3.	GSX		153
3.1	GSX in abstract		154
3.2.	GSX and devices		154
	3.2.1.	Device drivers and the ASSIGN.SYS file	157
3.3.	Using GSX		158
	3.3.1.	Setting up GSX commands	158
	3.3.2.	Calling GSX from BASIC	160
	3.3.3.	Installing GSX	163
3.4	A table of GSX commands and usage		164
	3.4.1.	Open GSX workstation	165

3.4.2.	Close GSX workstation	165
3.4.3.	Clear GSX workstation	166
3.4.4.	Update GSX workstation	166
3.4.5.	Place graphics cursor	166
3.4.6.	Remove last graphics cursor drawn	167
3.4.7.	Draw a polyline	167
3.4.8.	Draw a polymarker	167
3.4.9.	Draw text	168
3.4.10.	Draw a filled polygon	168
3.4.11.	Generalized drawing primitive (GDP) — BAR	168
3.4.12.	Set the height of a character	169
3.4.13.	Set polyline style	169
3.4.14.	Select polymarker type	170
3.4.15.	Set fill style	170
3.4.16.	Set fill index	171
3.4.17.	Set writing mode	171
3.5.	Pie Chart	172
3.5.1.	Initialise GSX system	173
3.5.2.	Call GSX	174
3.5.3.	Close GSX workstation	175
3.5.4.	Shut down GSX on error	175
3.5.5.	Open GSX workstation	175
3.5.6.	Update workstation	176
3.5.7.	Set style and index for polygon	177
3.5.8.	Draw pie slice	177
3.5.9.	Set the text height	180
3.5.10.	Draw text\$	180
3.5.11.	Draw labels	181
3.5.12.	Draw pie chart	183
3.5.13.	Data for graph	186
3.6.	BLOCK GRAPH	187
3.6.1.	Initialise GSX System	188
3.6.2.	GSX control routines	188
3.6.3.	GSX text routines	190
3.6.4.	Set PTS.IN for front/top/side	191
3.6.5.	Draw single block	192
3.6.6.	Draw a polygon	193
3.6.7.	Draw bar graph	194
3.6.8.	Data for graph	197
3.7	GSX TEST PROGRAM	199
3.7.1.	Initialise GSX system	200
3.7.2.	Various GSX control routines and error handl.	200
3.7.3.	Print data item and Data	201

3.7.4.	Print out GSX	202
3.7.5.	A specimen GSX__TEST output	207
4.	LOGO	209
4.1	Running LOGO	210
4.2.	Entering the programs	210
4.3.	L__GRAPH	211
4.3.1.	Linegraph	212
4.3.2.	Grid	214
4.3.3.	Do_grid	216
4.3.4.	Max	216
4.3.5.	Dograph	217
4.3.6.	Countchar	219
4.3.7.	Htext	220
4.3.8.	Line__test	220
4.4.	B__GRAPH	222
4.4.1.	Bargraph	223
4.4.2.	Dobar	223
4.4.3.	Bar__test	224
4.5.	ANIMALS	225
4.5.1.	List processing	226
4.6.	Lists and binary trees	228
4.6.1.	Getname	230
4.6.2.	Exist	230
4.6.3.	Savefile	231
4.6.4.	Loadfile	231
4.6.5.	Quit	231
4.6.6.	Cleardata	232
4.6.7.	Question	232
4.6.8.	First__answer	233
4.6.9.	Do__question	234
4.6.10	Is__ans	238
4.6.11.	Yes	239
4.6.12.	Do__ans	239
4.6.13.	Add__ans	240
4.6.14.	Setprompt	241
4.6.15.	Menu	242
4.6.16.	Some specimen data	244

INTRODUCTION

There is little doubt that in many countries, the Amstrad Personal Computer Word Processor range with its LocoScript software has opened up word processing to an entirely new generation of computer users. Where previous machines sold either to families looking for a new way of relaxing or to offices and professional users with a lot of money in their pockets, the PCW has reached out to thousands who simply want the advantages of the computer revolution, without the inflated prices that so often go with it.

Most PCWs are no doubt being used intensively as word processors and to users in this category we addressed a previous book, "Practical Amstrad Word Processing" (Sunshine Books, 1986). With time, however, many PCW owners are no doubt realising that word processing is not *all* that their machines are about. Quite simply, the PCW range represents a break through in cost-effective professional computer power which has not been seen since the introduction of the first low budget home computers. The aim of this book is to reach those who would like to extend their horizons to other low cost applications in the fields of finance, data handling and graphics.

The emphasis of the book throughout is practical. We are certainly not the only authors to be offering guidance on the use of the PCW, but we do have the advantage, as professional authors and software writers, of using a range of personal computers more intensively than the vast majority of private or commercial computer users. In our own work, computers are tools which must pay for themselves and we have brought that attitude to the PCW range in this book. During its course, readers will become acquainted with a variety of fresh uses for their systems, embodied in programs which have been developed over a period of years, versions of which are being used in countries all round the globe.

The material contained within the book falls into four main sections:

>> An introduction to the CP/M operating system, with the emphasis on the ways in which it can be used to configure the system more efficiently together with an outline of the more common operating procedures.

>> A collection of fully-documented programs in Mallard Basic covering various aspects of finance and data handling.

>> Details of the GSX graphics system supplied with the PCW and, for the first time in book form, how it can be practically used in conjunction with the power of Mallard Basic to enhance the output of Basic programs.

>> An introduction to the Logo language as it applies to business graphics and data processing by means of lists.

In writing the book we have tried to bear in mind that the majority of PCW users are not 'computer enthusiasts'. They do not relish hours of pecking away at the keyboard to unlock the secrets their machine conceals within its impassive case.

They just want to get on and use what they have bought. The object of the book is to help them do just that, as quickly as possible and using as many of the facilities available as possible. This is, in other words, a book not so much for reading as for *doing* and its aim is not so much to instill encyclopaedic knowledge but confidence in the use of the PCW.

At the same time, we have born in mind the difficulties that many owners have experienced in the use of the manuals supplied with the PCW and have included a detailed table of contents and index which should help to maintain the usefulness of the book in the long term. In addition you will find a number of tables which will be an invaluable aid in dealing with the more common CP/M and GSX functions. In one way or another we hope that this is a volume that will stay, well thumbed, on the desk for as long as the PCW itself.

CHAPTER 1

The CP/M Operating System

The intention of this chapter is to introduce some of the main features of the PCW's CP/M operating system. There are a variety of good books on CP/M available and it would not be possible to duplicate their contents in so few pages. Accordingly the main content of the chapter is a series of sections which show how the operating system can be configured to your own needs, saving time in daily use. An extensive action table gives a quick reference guide to the most important everyday tasks.

1.1. The operating system

No computer, no matter how powerful and sophisticated, can ever be better than the means of controlling it. The hardware of a machine may provide a host of facilities that seem close to science fiction, but none of them is the slightest use unless someone sitting at a keyboard is given some method of calling them up—an operating system.

The lowest level of the operating system is that of the 'bios' or 'basic input/output system', the complex program which is capable of giving simple instructions to the built-in hardware of the computer and of communicating with peripheral devices like disc drives, monitors, printers and so on. Without a bios, the hardware which goes to make up a computer is so much hi-tech junk.

The task of writing an operating system, however, does not end when the machine possesses a bios capable of controlling all the hardware. The fact that orders can now be given to the various devices is only one part of the equation; it remains to be determined what those orders are. In a personal computer, the answer is that the hardware needs to be ordered to perform the tasks set for the computer by the user. This is not necessarily a simple matter. It is one thing to build into the operating system the ability to control all the functions of a disc drive, to read a character of data from it or write one to it, to move the disc head around over the surface of the disc. Just doing that, however, will not load a program into memory from disc, copy a file from one place to

another or perform any of the host of functions that the user expects from a disc drive.

Having defined all the simple tasks that the hardware can perform, the next stage is to combine those simple tasks into more complex jobs which will be useful in the real world. It is at this stage, for instance, that the ability to read one or more characters from a disc, move the disc head and then write a number of characters, can be combined together to give the system the ability to copy a file from one place on a disc to another. The operating system of a personal computer like the PCW will be capable of literally hundreds of such tasks involving combinations of bios commands to the different devices that make up the system.

At this stage, it could be said that the operating system is finished. A properly written operating system at this level should be capable of running every kind of reasonable program written for the particular system and of performing all the tasks that the program requires. In order to achieve this the program will issue simple commands to the system, made up of combinations of characters which first flag the fact that a command is being issued, followed by others which indicate the number of the command.

Unfortunately the one factor that has been neglected in all of this is the user. The ability to call up all of the hidden power of the operating system by means of obscure combinations of characters is not much comfort to someone who sits down at a keyboard to set their system to work. While it may have once been true that computer users were content to sit at a machine and use only a program written for them by someone else, the advent of the personal computer means that users demand more control—even if only at the level of copying material from disc to disc, deleting items and so on. The tasks that the user wants to carry out are, as we have noted, already built into the operating system, so all that needs to be done is to add to the system a more amenable way for the user to call them up. The final layer of the operating system on most machines is therefore a program called the command line interpreter. It is this program which allows the user to type something like:

```
RENAME PROG1 = PROG2
```

at the keyboard and to see the command translated into action. With that, the operating system is regarded as complete.

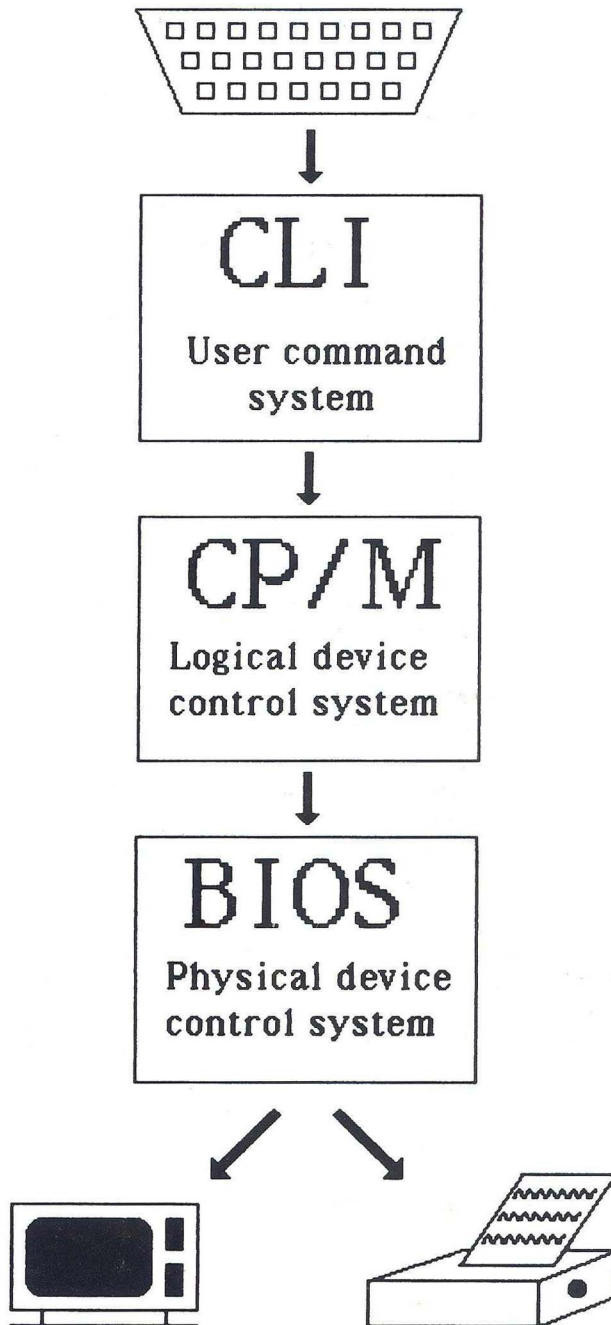


Fig. 1-1: A schematic view of the operating system

1.2. What is CP/M

The name CP/M represents a whole family of operating systems designed by Digital Research Inc., one of the first companies to provide serious software for micro-computers. What the letters CP/M stand for seems to be the subject of some disagreement, since it seems that no-one can remember whether in the earliest days they were meant to stand for 'Control Program for Microcomputers' or 'Control Program and Monitor'. Whatever the name, the original CP/M was written to allow users of the earliest micros to control their systems, and especially their disc drives, in a standardised way, as compared to the vast range of operating systems that ran on the computers manufactured by the big mainframe companies.

CP/M rapidly became the industry standard, the yardstick against which every operating system was measured. More and more programs were written using the standardised calls which CP/M provided for access to discs and other system features. Buyers of a new micro which ran CP/M did not have to wait years for software to be written for it, they had access to the vast library of CP/M based programs from the very start. Over the years since the early 1970s, CP/M has changed to meet the demands of successive generations of chips and micros, but the aim is still to provide the user with a wide range of controls over the hardware that makes up a system using, as far as possible, standard commands.

1.2.1. CP/M Plus

CP/M Plus, the operating system chosen for the PCW range, is vastly superior to the early forms of CP/M. It is specifically adapted to the PCW's need to be able to regard its memory as a series of blocks, so that the maximum 64K of memory which can be handled at any one time by the Z80 chip running the system, can be exceeded. It is the CP/M Plus operating system which permits your PCW to run programs which on other machines would be beyond the capabilities of the Z80 chip. In addition to this major step forward, in the new version most of the traditional commands have been updated and new facilities added—like built in commands which do not require access to files on disc.

1.3. Some CP/M concepts

To make the most of your CP/M system you need not only to understand the major utilities it offers, but the way in which it structures your system. In the sections which follow we shall take a brief look at the idea of physical and logical disc drives, physical and logical devices, user groups and the ramdisc.

1.3.1. Physical and logical disc drives

The number of disc drives you will have on your system will depend upon the model you are using. The PCW 8256 comes equipped with a single disc drive capable of using one side at a time of Amstrad's CF2 discs. The PCW 8512 has a second disc drive of much higher capacity, which is capable of accessing both sides of the disc without having to be turned over.

On a two drive system, the drives are referred to as A: and B: and at any one moment, one of them will be regarded as the 'default drive'. When the system is told to carry out an operation involving a disc file of any kind, unless the file name is preceded by the drive specifier, only files on the default drive will be used. The default drive can be changed by entering A: or B: on the command line in CP/M and which of the two discs is the default at any one time will be displayed by the command line prompt, which will either be A> or B>.

If you do not have a second drive fitted to your machine it is important to remember that you still have access to two 'logical disc drives'. What this means is that you can make the system treat the disc drive as if it were somehow split in two. Entering A: or B: to change the drive, on a single drive system, will bring up a message asking to you to place the A: (or B:) disc in the drive and then press a key. Once this has been done, the system will treat the newly inserted disc as if it were in another drive to the one which has just been removed.

The advantage of this is that it allows the single drive user to employ twin-drive techniques, like copying files from disc to disc. If you start with the A: drive as the default and enter a copy command like:

```
PIP B:=A:FILE1
```

the result will be that the file called FILE1 will be copied from the current disc to a second disc which you place into the drive when prompted to insert the B: disc.

1.3.2. Physical and logical devices

The distinction between physical and logical does not only apply to users of the single drive systems but to a wide range of applications within the system. As with discs, where what the user works with is not so much the actual disc

drive as the logical disc drive, so most of the work of the system is done by logical devices—names given to actions which the system can carry out. Of course the actions are really carried out by ‘physical devices’ but which physical device is attached to which logical device can to a large extent be determined by the user.

An example may help to make this clearer. Two of the physical devices on your system are LPT:, which is the name of the physical device which is the printer, and CRT: which is the monitor and keyboard. CONOUT: (console out) is the name of the logical device to which the output of the system is normally sent. In normal circumstances the physical device CRT: is attached to CONOUT: so that when the system sends its output to CONOUT: you see it on the screen.

If you would like to demonstrate that things do not necessarily have to be like this, put some paper into your printer, insert a copy of a disc with the file DEVICE.COM into the current default drive and then enter the following on the CP/M command line:

```
DEVICE CONOUT: = LPT
```

This tells the system to connect the logical device CONOUT: to the physical device LPT:. Your printer should immediately start up, and display a list of logical devices and the physical devices attached to them. You will also find that from now on, all the output which would normally go to the screen is output through the printer. To get back to output through the screen you must either carefully enter:

```
DEVICE CONOUT: = CRT
```

or reset the system.

(Note: This demonstration is only intended for those using the basic system with the standard printer supplied. Those who have added a serial or parallel printer via the CPS8256 interface would need to replace LPT in the first command with the name of the physical device (CEN or SIO) associated with their printer.)

All of this is not merely a matter of theory. Changing the way devices are allocated is an important part of configuring the system. If you wish to use the optional interface CPS8256 to add a different printer to your system, you would have to tell the system that the logical device LST (list device), which is where it sends all output which it expects to be printed, is now connected to the physical device CEN or SIO representing the two ways (centronics and serial) that the interface has of communicating with the outside world.

1.3.3. User groups

The idea of user groups is one that seems to have some people rather confused, which is a shame because in many ways they are one of the nicest features of the PCW.

The object of user groups is to help you sort out your material. Without groups it would be quite possible to build up discs with such a mass of files on them that it would become very difficult to find anything or to name things so that you could identify them at a later date. If you wish to use several different programs on the same disc then user groups will allow you to separate them and their material into different sections of the disc where they will not conflict with each other. This is the way that LocoScript uses the groups, since file groups in LocoScript are simply CP/M's user groups given a new name.

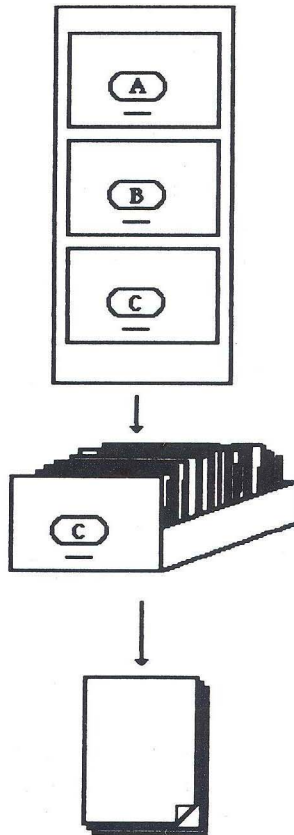


Fig. 1-2: One way of looking at groups

The illustration is one way of attempting to explain what user groups are about. What it tries to show is that user groups are a way of separating files into separate compartments on a disc. The illustration is based on the idea that a single disc is like a filing cabinet. It contains a great deal of information, but that information is not all lumped together in one mass, it is divided into groups which, in the case of the filing cabinet, are called drawers. A single drawer may contain a great many files or only a few. Provided we know which drawer contains which kind of material, any single file is a great deal easier to find.

There are sixteen different user groups on a disc, numbered from zero to fifteen and they are always there, regardless of whether you place anything into them. When you first start up your system you are actually using user group zero, though there is nothing about the prompt `A>` to remind you of this. If you change to any other group, however, using a command like:

USER 15

The prompt will change to reflect the group you are in, ie `15A>`. Only in group zero is the number not present.

In addition to changing the user group on the present disc, it is important remember that the **USER** command changes the user number for the whole system, so that the user group on any other drive will be changed. If this seems strange it is because in many ways CP/M Plus is a stripped-down version of a multi-user operating system, where up to 16 people at a time could have access to the system, all with their own separate compartments on the discs in the various drives. The system as you have it does not permit separate users but it still works on the assumption that user number `X` should only have access to group `X` on any discs in the drives.

Once you changed to a particular group you are limited to the use of material and programs which are currently in that group. With one important exception you cannot run a program which is in another group unless you first change into that group. Equally, a program running in the current group cannot gain access to material in another group if it is needed. You can, however, copy material from group to group using the command given in the action table later in the chapter.

The only exception to all of this are system files, that is, files which have a special marker known as the 'system attribute' set (see Section 1.4.1.1). Files which run commands, like **DIR**, **RENAME** and **PIP** often have the system attribute set, but it can be set for any file on the disc. In actual use there is

only any point in setting the system attribute on files with the terminators .COM and .SUB which contain programs which can be executed—you *can* set the system attribute on a text file if you wish but it will make no difference to its use since you cannot ‘run’ a text file.

The importance of the system attribute is that whenever a command is entered, if the program it refers to cannot be found in the current user group, the system will then scan through user group zero to see if there is a system file of the correct name in that group. If so, the program it contains will be executed. This gives you an extra range of flexibility in the use of discs, since if a program like BASIC.COM is installed as a system file in user group zero, using the techniques described in Section 1.5, you will have 15 remaining groups which can hold different types of your own BASIC programs, all of which can be run and edited using the same BASIC.COM program.

1.3.4. The ramdisc

Before moving on to some of the practicalities of using your CP/M system, one last subject needs to be tackled, and that is the ramdisc or ‘memory drive’, as it is often called on the PCW range. The ramdisc is a powerful feature of the PCW which makes it both more convenient and considerably faster to use than some comparable machines sold at far higher prices.

The essence of the ramdisc is that it is an extra disc drive with a useful capacity of around 112,000 characters (larger on the 8512). You can store files on the ramdisc in the same way that you can on disc and perform all the operations on them that you can on a disc file. The only difference is that a file stored on the ramdisc only exists inside the RAM (Read And write Memory) of your PCW, and only for as long as the machine is switched on. The moment that you switch off the computer, reset it using SHIFT/EXTRA/EXIT, or there is momentary failure in the power supply, everything stored on the ramdisc will be lost completely. For this reason we would recommend that you never store the only copy of important files on the ramdisc alone.

The advantage of the ramdisc is that it is very fast to use when accessing files and that it allows you to keep often used bits and pieces ready at hand, since changing discs will not affect the contents of the drive. One common way to use the ramdisc is to set up your system so that when it is switched on or reset, common CP/M utilities like DIR and PIP are copied onto it (see Section 1.4.1.1).

Other than the fact that material is lost when the machine is reset, the ramdisc behaves like a normal drive. Its drive specifier is M: and you can use it as the default drive, change user groups, copy from it or to it, run programs from it, all the things you would expect to be able to do with a normal disc drive.

1.4. Structuring a disc for CP/M

One of the first tasks that you need to undertake with your system is to set up some discs which will speed the process of using CP/M and the programs that you can run under it. We shall not deal with the techniques of structuring discs for use with the LocoScript package. This is a specialised area which is described in our previous book, *Practical Amstrad Word Processing*.

The first thing to remember when using CP/M, or indeed any other commercial software, is that you do not use the original discs supplied but take copies of them using the DISCKIT program. This ensures that if any damage is ever done to the disc with which you are working, you will not be left with a non-functioning computer until you can purchase a replacement. Having completed the copying you will have a set of discs containing all the CP/M operating system, the utilities that go with it and the various other free programs supplied with the system, including the LocoScript package.

As far as CP/M is concerned, having made your copy the problem still remains that while the material making up the basic operating system is essential to everything you will do, most of the files on the main CP/M disc will be employed only occasionally. In daily use, a disc with the complete set of CP/M system and utility files would be a hindrance, since a great deal of space would be taken up which would be better used in storing programs and files which *are* in regular use.

For this reason, we recommend that you set up two different kinds of discs. The first kind will be used to configure the system when it is switched on or reset. The second will store the programs and material which you wish to use. Later on in the chapter we shall see that there is a third kind of disc which will allow you to run important applications automatically.

1.4.1. Creating a start-up disc

The purpose of the start-up disc described in this section is to allow you to start or reset your system, configure it to your needs, transfer a number of commonly used utilities to the ramdisc and then remove the disc and replace

it with a work disc dedicated to the kind of activity you want to undertake. In this way you will have a great deal more disc space available than if you tried to work with on a disc containing the CP/M system and all its utilities.

The procedure for creating a start-up disc is as follows (users of two drive systems may adapt the instructions to allow for direct disc to disc copying, but this method will work for either system):

>> Use the DISCKIT program to take a copy of the main CP/M disc and label the new disc CP/M START-UP.

>> From the START-UP disc, delete the file BASIC.COM, using the command:

ERA BASIC.COM

This removes from the disc the program which runs Basic and makes space for a number of other useful utilities. The Basic interpreter will later be given a disc of its own.

>> Insert in your master copy of side three of the original discs supplied with the system (if for any reason the configuration of the discs has been changed, simply find the disc or discs holding DEVICE.COM and DATE.COM).

>> Enter the commands:

PIP M: = A:DEVICE.COM

PIP M: = A:DATE.COM

This use the PIP (or Peripheral Interchange Program) to copy to the ramdisc the CP/M utility which allows different physical devices to be associated with the logical devices recognised by the system and another which allows you to set a date which is remembered by the system.

>> Replace the START-UP disc and enter:

PIP A: = M:DEVICE.COM

PIP A: = M:DATE.COM

This copies the programs to the START-UP disc.

>> You now have a working CP/M disc with a few more utilities. There

remains only the job of making it into a real start-up disc by adding a 'submit file' which will configure the system when it is switched on or reset.

1.4.1.1. *Creating a PROFILE.SUB file*

Whenever your PCW system is switched on or reset, the first thing it does after loading the CP/M system into memory is to look for a file called PROFILE.SUB. This is what is known as a submit file and consists of a series of CP/M commands which are carried out automatically whenever the file is run, rather than having to type each one individually.

It is possible to create submit files for a variety of purposes—what marks out PROFILE.SUB is that it is always the first program run by the system (provided that the program SUBMIT.COM is on the START-UP disc). The importance of this is that if you wish to configure your system in a particular way before you begin working, the commands to accomplish it can be placed in PROFILE.SUB.

The software delivered with the PCW range normally includes a file called PROFILE.ENG, which from its name appears to be a version of PROFILE.SUB used during the development of the system. The file you are going to enter here is more adapted to everyday use.

To set up the file you will need to make use of the special text editing program supplied with CP/M called ED. The procedure is as follows:

>> With your START-UP disc in the drive, type:

ED PROFILE.SUB

This calls up the ED program and tells it to edit a file called PROFILE.SUB.

>> Since the file does not exist, ED places a message on the screen telling you that this is a new file and then waits for you to tell it what to do. Given below is a list of the lines you must enter. In each case, the text to be entered is what you see to the right of the colon (:)—ED supplies the rest. At the end of each line, press RETURN. The only exception to this is on line 16, when the file is finished and you should press the EXIT key, followed by 'e' and RETURN, which stores the file you have just entered on the disc and takes you back to the operating system. Details of the ED program and its usage are contained in the advanced programming tools section of the CP/M manual supplied with your machine, but they are not necessary if you simply follow

the illustration laid out below.

>> The lines to be entered are as follows:

```
A>
A>
A>
A>
A>
A>
A>
A>ed profile.sub
NEW FILE
: *i
1: setdef m:,a: [order=(sub,com)]
2: pip
3: {m:=dir.com[o]
4: {m:=erase.com[o]
5: {m:=rename.com[o]
6: {m:=submit.com[o]
7: {m:=set.com[o]
8: {m:=set24x80.com[o]
9: {m:=show.com[o]
10: {m:=type.com[o]
11: {m:=setkeys.com[o]
12: {m:=pip.com[o]
13: {m:=date.com[o]
14: {
15: set m:*.com[sys]
16:
: *e
```

A)■

Drive is A:

Fig. 1-3: The PROFILE.SUB file as entered with ED.

>> The effect of the PROFILE.SUB you have entered is as follows:

SETDEF M:,A: [ORDER=(SUB,COM)]—Tells the system the order in which it is to search its disc drives when asked to run a program—in this case it is told to look first on the M: drive, then on A: drive. In each case, the program it will look for will be one with the name entered on the command line by the user. If, however there are two files with the same name, one with the terminator .SUB and one with .COM, the instruction tells the system to run the submit file rather than the main .COM file. The advantage of this is that in the case of many programs, it can be useful to create a submit file to set up some aspect of the system before calling the main program. Setting ORDER in this way ensures that your submit file will always be carried out first.

(Note: Users with two-drive systems will wish to insert the name of their second drive at the end of the list of drives to be searched.)

PIP—The set of commands entered under PIP copy a series of often used utilities to the M: drive, where they will always be available for use, no matter how often the discs are changed.

SET—Having copied the CP/M utilities to the ramdisc, they all have the system attribute set, so that they can be accessed from any user group. The asterisk in this command is a 'wild-card' which tells the system to apply the command to any file.

>> Having created the PROFILE.SUB file on the START-UP disc, reset the system using the SHIFT/EXTRA/EXIT keys. Once the CP/M system has loaded, you will see the commands contained in the submit file being carried out. The result will be a system which provides a great deal more flexibility than if you were forced to keep a CP/M disc in the drive at all times in order to make any use of the utilities. All the utilities you have copied to the M: drive are now continuously available.

>> To demonstrate the value of what you have done, take the START-UP disc out of the drive and enter DIR M: [FULL]. A directory of the M: drive will appear, despite the fact that your CP/M disc is absent.

1.4.1.2. Customising PROFILE.SUB

Different systems have different needs, according to the way in which they are used. In many cases, the distinctive features of a system can conveniently be set up in the PROFILE.SUB program rather than the commands being entered manually every time the system is used. Some of the more common uses of the PROFILE.SUB file in configuring the system are:

a) Adding new devices to the system using the DEVICE command. An example of this, for those who have added the Amstrad CPS8256 interface, would be to add the centronics port to the system as the main printer port:

```
DEVICE LST: = CEN
```

b) Setting the parameters for the serial port if you have added the CPS8256 interface, as in setting the baud rate of the interface to 300:

```
SETSIO TX 300 RX 300
```

c) Configuring the keyboard to provide special characters on particular keys using a keyboard file and the SETKEYS utility.

d) Sending an an initial command to a device (say a printer) using the SETLST command, for instance to set the printer into letter quality mode. Details of the initialisation of a printer are given in the CP/M manual under SETLST.

e) Setting the screen to 24*80 using the command:

```
SET24X80 ON
```

f) Starting up the system in a particular user number, as in:

```
USER 11
```

Any configuration or action which can be achieved using CP/M commands can be written into the PROFILE.SUB file if you think it will save you time in the long run. To add a command to the file, use the ED program or the free RPED editor, which runs in Basic.

1.5. Working discs

Having created a START-UP disc, the next task is to take a look at the techniques of setting up a working disc to use with a utility program of some kind. In this section we shall assume that the utility you are going to use is the Basic interpreter program, BASIC.COM. The same techniques will apply to the vast majority of utilities you can use on your system.

The basis of a working disc is that the programs necessary to run the utility in question are all placed into user group zero and each program has its system attribute set. You are then free to store the material on which the programs work either in user group zero or anywhere else on the disc, since group zero programs with the system attribute set, have access to all 16 of the user groups.

The procedure for creating a Basic working disc is as follows:

>> Restart the system using your START-UP disc.

>> Prepare a freshly formatted disc. For a single drive system, the disc should be formatted for Drive A:. Dual drive system owners should prepare a disc for the B: drive.

>> Place your original copy of the CP/M disc containing BASIC.COM into the A: drive and enter:

```
PIP B: = A:BASIC.COM
```


If you have dual drives the process will be carried out automatically. Single drive users will have to follow the prompts, treating the freshly formatted disc as disc B:.

>> In the case of some other utilities or commercial programs, the overall package may consist of several programs or other files containing important material—the Logo language described in Chapter 4 is an example. In such cases you must read the documentation for the package and copy all the necessary files to the working disc.

>> Set the system attribute for all the files in the user group (in this case there is only one) by entering:

```
SET *.*|sys,ro|
```

The importance of this is that it not only sets the system attribute for each file but also makes the files 'read only' so that they cannot be accidentally erased, except by formatting the disc. If you do wish at some future date to change a file to 'read/write' status so that it can be erased, enter:

```
SET <filename>|rw|
```

>> Enter DIR and you will find that no files are listed but that system files exist in the group. It is important to remember that files with the system attribute set do not appear in the normal directory. To see them you need to enter DIRSYS or DIR[FULL].

>> Having copied BASIC.COM, with the working disc in the drive, enter:

```
USER 1
```

>> Now enter:

```
BASIC
```

>> The screen will now display the Basic start-up message and you have access to all of Basic's facilities.

>> Enter a line of Basic, as follows:

```
1 REM THIS IS A BASIC PROGRAM IN USER GROUP 1
```

>> Now enter:

SAVE "TEST",A

>> Once the disc drive has stopped, enter:

SYSTEM

and you will return to CP/M.

>> Enter:

DIR

and you will find that in the current group, group one, the Basic program TEST has been created.

>> You are now free to create Basic programs in any of the different user groups, separating programs into different groups according to their function. The same, of course, applies to any utility program

>> One hint when using properly structured discs like this, is that you can list out the whole of the contents of the disc on the screen, thus saving you the trouble of changing into each group in turn, by entering the command:

DIR [USER = ALL]

The list can be sent to the printer by pressing ALT/P before pressing RETURN to enter the command and again after the list has finished (see the list of control keys in Table).

1.6. Protecting files

We have already touched on this topic in the previous section, but it is worth highlighting the fact that some files are worth preserving from corruption or deletion by protecting them. This process is carried out using the SET command, as follows:

SET <filename> [RO]

The 'RO' stands for 'read-only' and the effect of the command is that apart from formatting the disc, the system will refuse to allow any change to be made to the file in question, including its deletion from the disc. This is an

important feature of the system when discs are being used intensively, since it is very easy to accidentally erase a file by entering a name incorrectly.

To change the status of a read-only file back to read/write, so that it can be modified or deleted, enter:

```
SET <filename> [RW]
```

1.7. Single purpose discs

In many cases, the number of packages which will be run on the system is relatively few and the easiest way to use them will be to install them on single purpose discs. Users of the PCW are already familiar with the idea of the single purpose disc since it is such a disc which runs LocoScript. A single purpose disc is simply a disc which, when it is placed into the disc drive and the system started up or reset, runs the package to which the disc is dedicated.

A single purpose disc contains the CP/M operating system but, in the vast majority of cases, none of the CP/M utilities except SUBMIT.COM. There will be a PROFILE.SUB file but it will normally consist of a single command to run the desired package. Given below are instructions for creating a single purpose Basic disc. As with the working discs described above, the techniques will apply to most packages:

>> Copy the CP/M disc onto a fresh disc.

>> Using the ERA command, delete all the files from the disc except:

a) SUBMIT.COM, which will be used to run the PROFILE.SUB file.

b) Any files which have the terminator .EMS because these contain the CP/M operating system.

c) Any CP/M utilities which, according to the instructions supplied with the package, are used by the package itself—there are none in the case of Basic.

d) The ED.COM file, which will be used to create PROFILE.SUB.

>> Copy the BASIC.COM program onto the disc (for other packages, copy all the relevant files) and set the system and read only attributes.

A)

A)

A) ☐ B) ☐ C) ☐ D) ☐ E) ☐

H2
 A3

03

02

11/11/2011 11:11:11 AM

À

A)

11

11

Notes:

File = file name—wild cards *not* allowed.

File(s) as above but wild cards *are* allowed.

{ } = denotes parameters are optional.

Action	Command	Files
Allow files to be deleted	SET <file(s)> [RW]	SET.COM
Change logged disc drive	<drive letter>:	None
Change user number	USER <new user>	None
Copy a complete disc	DISCKIT	DISCKIT.COM
Copy between user groups	PIP A: = A: <files> [G<num>]	PIP.COM
Copy dir file(s) to disc	PIP B: = A: <file(s)>	PIP.COM
Copy dir file(s) to ramdisc	PIP M: = A: <file(s)>	PIP.COM
Copy on default drive	PIP <new file> = <file>	PIP.COM
Copy sys file(s) to disc	PIP B: = A: <file(s)> [OR]	PIP.COM
Copy sys file(s) to ramdisc	PIP M: = A: <file(s)> [OR]	PIP.COM
Erase file(s)	ERA <file(s)>	ERASE.COM
Examine a text file on screen	TYPE <file(s)>	TYPE.COM
Examine any file on screen	DUMP <file>	DUMP.COM
Format a disc	DISCKIT	DISCKIT.COM
Get help on commands	HELP	HELP.COM HELP.HLP
Make GBASIC (see Chapter 3)	REN GBASIC.COM = BASIC.COM GENGRAF BASIC.COM	RENAME.COM BASIC.COM GSX.SYS GENGRAF.COM
Print text file	CTRL P TYPE <file(s)> CTRL P	TYPE.COM

Print text file, with numbered lines	PIP LST: = <file>	PIP.COM
Protect files from deletion	SET <file(s)> [RO]	SET.COM
Rename file(s)	REN <oldfile> = <newfile>	RENAME.COM
Run BASIC BASIC	BASIC.COM	
Run GBASIC	GBASIC	GBASIC.COM ASSIGN.SYS DDSCREEN.PRL L DDFXLR8.PRL DDFXHR8.PRL
Run Logo	LOGO	LOGO.COM SUBMIT.COM LOGO.SUB KEYS.DRL SETKEYS.COM
Run submit file (at any time)	SUBMIT <file> [<pars>]	SUBMIT.COM
Run submit files (ONLY after SETDEF has set submit files in ORDER)	<file> [<pars>]	SUBMIT.COM
Send control sequence to LST	SETLST <file>	SETLST.COM
Set new keyboard	SETKEYS <file>	SETKEYS.COM
Set screen size to 80 chars.	SET24X80 [ON]	SET24X80.COM
Set screen size to 90 chars.	SET24X80 OFF	SET24X80.COM
Set search path for command	SETDEF <disc drive:>	SETDEF
Set SIO parameters	SETSIO <parameters>	SETSIO.COM
Set SUBMIT file execution (Avoids having to type SUBMIT <file>)	SETDEF [ORDER = (SUB,COM)]	SETDEF
Verify a disc copy	DISCKIT	DISCKIT.COM
View all directory files	DIR [<file(s)>]	None
View all system files	DIRSYS [<file(s)>]	None
View full files information	DIR [FULL][<file(s)>]	DIR.COM

1.9. Table of recommended CP/M file extensions.

When examining the contents of discs supplied with the system or when creating files of your own, you may find it useful to refer to this table which lays out the conventional terminators given to files serving different purposes on the system. In some cases, the terminators are merely a matter of clarity, helping to distinguish between different file types. In other cases, altering the file terminator will result in either the system or an individual program not recognising the file and therefore being unable to make use of it.

ASM—Assembler source code

BAS—Basic program

BAK—Backup file (old copy of file)

COM—Executable machine code program

DAT—Data file

DTA—Data file

EMS—The CP/M operating system.

IDX—An index file from a database.

KEY—A keyboard file.

LOG—Logo source code

LST—A listing file, from a language compiler or assembler.

OBJ—Assembler object code

PRL—GSX device driver

SUB—Submit file

SYM—A file containing symbols, from a language compiler or assembler.

SYS—'System' file—not to be confused with a file with SYS attribute.

TXT—An ASCII text file

\$\$\$—A temporary file.

1.10. CP/M editing keys

When using CP/M and some CP/M based programs, a number of commands and quick editing facilities are available, which permit short-cuts in the entry and alteration of command lines, together with a number of extra facilities. The following table lists out the commands available:

Keys	Purpose
ALT-A	Equivalent to the cursor left key.
ALT-B	Moves the cursor to either the beginning or end of the command, depending on its present position.
ALT-C	Interrupts the current program—equivalent to the STOP key.
ALT-E	Breaks the command being entered into two on the screen with a carriage return, without altering the meaning of the command.
ALT-F	Equivalent to the cursor right key.
ALT-G	Deletes the character under the cursor—equivalent to DEL→ key.
ALT-I	Equivalent to the TAB key.
ALT-H	Deletes the character to the left of the cursor— equivalent to the ←DEL key.
ALT-J	Equivalent to pressing RETURN. In some programs will move the cursor down without actually creating a carriage return.
ALT-K	Deletes all characters from the cursor position to the right hand end of line.
ALT-M	Equivalent to RETURN key.
ALT-P	Switches on and off the facility which copies screen output to the printer—particularly useful in obtaining copies of directories and output of other CP/M utilities.
ALT-Q	Enables screen output (see ALT-S).
ALT-R	Copies characters to the left of the cursor onto a fresh comand line—the characters to the right of the cursor are forgotten (see ALT-K).
ALT-S	Freezes screen output. If the program is outputting to the screen it too will be frozen (see ALT-Q).

- ALT-U** Cancels the current command string and moves the cursor to the next line—ie, the command is not sent to CP/M (see ALT-X). The characters to the left of the cursor are copied into a special memory area called the command line buffer and can be recalled (see ALT-W).
- ALT-W** Recalls to the screen the last command entered (ie, not deleted or abandoned) if the command line is empty. If the command line is not empty the cursor will move to the end of the line. This key combination is particularly useful when repeating commands with small changes.
- ALT-X** Deletes and forgets all characters to the left of the cursor—not the character under the cursor.

On the PCW system some specific key combinations have been added, the more important of which are as follows:

- COPY** Equivalent to ALT-W
- CUT** Equivalent to ALT-U
- EXTRA
PTR** Creates a screen dump on the printer.
- PASTE** Equivalent to ALT-W
- PTR** Puts the system into printer control mode.
- RELAY** Equivalent to ALT-R
- SHIFT
EXTRA
EXIT** Resets the machine.
- STOP** Equivalent to ALT-C above
- F3/F4** Equivalent to ALT-Q
- F5/F6** Equivalent to ALT-S
- F7/F8** Equivalent to ALT-P

CHAPTER 2

Basic

It is a common myth that the letters Basic stand for “Beginners’ All-purpose Symbolic Instruction Code” invented in the 1970’s at Dartmouth College to simplify the process of teaching people to program computers.

Certainly it is true that *a* language was invented and given the name Basic, but it long ago became quite ludicrous to go on pretending that there was one animal called Basic. There are Basics that incorporate features of almost every other conceivable computer language—Basics that behave very like Logo, Basics based on the language ‘C’, Basics that do almost nothing and Basics with facilities to make your mind boggle. Whenever someone criticises Basic on the basis that it lacks X,Y or Z compared to other computer languages, it is almost inevitable that a new form of the language will be introduced containing X,Y,Z and a few extras besides.

Basic now means “the language that most people use” and as inadequacies are pointed out, the language that people use develops to eradicate them. Many computer snobs delight in predicting the end of Basic and its replacement with just about every new language that has been developed. They are wrong, and they will continue to be wrong, for the simple reason that they are shooting at a target that is and always will be moving too fast.

The other failing that many people have when it comes to Basic is a lack of appreciation of its power. Once again, computer snobs are prone to argue that nothing complex can be done in Basic—it is too clumsy, too slow, too almost anything. This, of course, doesn’t stop hundreds of commercial programs being developed in Basic because of the ease with which it is written, before the program is compiled, or translated into the form of “machine code” appropriate the particular chip the program is to be run on. Even for uncompiled Basic, however, the question of speed is often comparative. Many simple applications which could benefit from the use of a computer can just as easily be programmed in Basic provided that you do not mind waiting 1 second for the result rather than the 0.1 or even 0.001 of a second needed by a program written or translated into machine code. Those with large quantities of data to process will be happy to pay the price of specialist programs but

many computer owners will not and there is no reason why they should be deprived of the benefits that programming in Basic can bring.

2.1. Basic and the PCW

There is little question that the vast majority of PCW owners buy their machines in order to use the LocoScript word-processing package and other commercially available programs for a variety of purposes. Sadly, many of them never go on to realise that they have also purchased a powerful tool which, with a little thought and preparation could be used to carry out specific tasks of their own, without the need to buy expensive commercial software.

The PCW provides those who do wish to plan and write their own programs with a number of free facilities. The Logo language is supplied with the machines, as is an “assembler” which allows programs to be written in “assembly language” by those who have the expertise. For most people, however, the most flexible facility provided for programming the system will be the excellent Mallard Basic which is provided with the PCW.

Mallard Basic is an extremely powerful language which bears little relation to the versions of Basic available on the average home computer. As a language it is designed not for the playing of games or the creation of interesting sound and graphics but to the handling of information in a variety of forms—the task which microcomputers do best.

Perhaps the most significant advance represented by Mallard Basic is the advanced facilities it contains for handling data files on disc. Those who have created data handling programs of their own will know how difficult it can be to manage on disc quantities of data larger than the memory of the computer. In Mallard Basic this problem is largely overcome by means of the “Jetsam” commands which take over the ordering and indexing of a file on the disc, allowing the programmer to save, search for, examine or retrieve items from the file with great ease.

In the programs which follow in this chapter, you will find yourself presented with a variety of methods of handling different kinds of data. For many readers, the majority of programs can probably be applied directly to their own immediate needs. For others, the ideas may be useful but adaptations may need to be made. Whichever is the case, the programs are a resource which the reader can use over a period of time to apply the power of the PCW to their own situation. In using them it is important remember that they are not

offered as holy writ. Where they can be improved or tailored to special situations, the changes should be made. Where they point out the usefulness of the system in a particular sphere but are gradually outgrown, they can be replaced with commercial software which, unlike Basic, can make use of the whole of the memory of the PCW.

2.2. Entering the Basic programs

In this book you will find that all of the programs are constructed out of clearly identifiable modules. The reason for this is that programs written in modules can be more easily read, they can be more easily debugged, they can be changed by substituting modules which work more efficiently if you learn new methods, they can be added to by patching more modules in. There is much to be learned from the programs which follow but probably the most valuable lesson of all for your future programming will be the technique of modular programming.

In entering the programs you will be invited to type them in module by module, observing as you do so the commentary on the lines of the module. The order in which the modules are entered is not the order in which they will be found in the program when it is complete. The order of modules in the overall program is determined by factors like the need to make the overall program readable, the desire to place frequently used modules towards the beginning of the program and the order in which material came into the authors' heads.

As you work on each program it is impossible to stress too highly the importance of saving your work at regular intervals so that if an accident should occur, you will not find yourself having wasted hours of work. To emphasise the importance of this, a simple method of encouraging yourself to save your programs regularly is given in the first module of the first program.

2.3. Testing programs

The order in which the book presents the modules is designed so that in most cases it should be possible for you to conduct simple tests for yourself of the main functioning of a module. For instance, if a module which displays a menu on the screen relies on another module to position a title, the title module will normally be entered first, followed by the menu module. The

advantage is that if you wish to then test the menu module by running it, even though the majority of the program is not present, it should at least provide you with the correct screen display without locking up when it cannot find the title module.

We recommend that as you go through the programs you do pause often to save the program and then run the particular modules you have entered or the whole program so far. In many cases this will produce error messages as the system informs you that certain lines have not been entered yet, variables not defined and so on. In other cases the program will appear to run quite normally, it will simply be without a number of facilities which you will not have entered yet. The only errors which can never be allowed to pass are the notification that you have a 'syntax error' or 'type mismatch' in one of the lines. This means that the system cannot understand the line and, since there are no such errors in the listings provided, it also means that somewhere the specified line is incorrect.

In cases where what you have entered of the program so far runs without hitch, it is often wise to make use of the modules which save and load data from disc to store some specimen data. Subsequent tests will then be easier, since rather than having to enter further test data manually, you can simply recall it from disc.

When you have finished entering each program, test it. One way of doing this is to use the testing procedure given below, which we have found to be useful in highlighting problems in data handling programs. The standardised procedure is as follows:

- 1) Enter a single item of data appropriate to the program and then display it by whatever means the program provides. Any problems at this stage will normally be either in the input module of the program or in the module which displays material. Which of the two is at fault can usually be determined by printing out the contents of the first element (element zero) of the arrays used by the program. If the data appears to be sensibly stored when compared with the description of program function in the commentary, the fault is probably with the output module.

- 2) Enter another four items of data. If the program is designed to sort items, enter the four items in the reverse order to the one in which they should be sorted. Page through the items using the search/display module to ensure that they are in the correct order and that the program does not allow you to move off the end of the data. Problems at this stage are likely to be with the sorting module or, again, because the output module cannot handle multiple

items—you can discover whether the material is stored correctly by printing out the contents of the first five elements of the arrays used.

3) If the data is stored sensibly, use the data saving module to store it on disc.

4) Using whatever facilities the program provides to delete items, delete the first item in the file. Examine the remaining items to ensure that the program is capable of deletion and that it moves the remaining items properly down the array.

5) Delete what is now the fourth item and the second item, examining the data after each deletion.

6) Now delete the remaining two items and see that the program behaves sensibly when it has no data to work on.

7) Reload the data which you saved on disc and go through the process again from step four. This ensures that the data stored on disc has not been corrupted in some subtle way.

8) Having tested the gross functioning of the program, go through the commentary on what you have entered and make specific checks on the functioning of each module, where possible.

9) If all these tests are successful then it is highly likely that the program is working fully, though it is always possible that in use, minor faults will show up which may have to be rectified.

2.4. Banker

The object of this first program is to allow the user to keep a clear and continuously updated record of a single bank account, the names of payments, their date and the amount, including the ability to specify not only single payments, but recurring expenses or receipts, no matter how irregular the period. The program is designed to deal with an account for the period of one calendar year, though if more than one thousand transactions are involved, limitations of memory may mean that you will need to start afresh with a new record every few months.

Statement:

JANUARY		
DAY & DETAILS	ITEM	BALANCE
Balance forward		0.00
1 Salary	825.60	825.60

Press a key ...



Drive is A:

Fig. 2-1: A typical account compiled by Banker

2.4.1. Saving the program

These four lines may seem a trivial place to start, but those who have already used this kind of technique will know that this little module can save an immense amount of heartache in the development of programs.

Most people only learn by bitter experience that programs *must* be saved regularly as they are developed. Sooner or later most of us run up against the time when hours of work is thrown away because of a momentary surge in the power supply, a blown fuse, a knock to the micro or a plug. Experienced users will have lost only some fifteen minutes work because they will never have allowed more than that time to pass without saving the program thus far entered.

The purpose of the four lines in this module is to encourage you to make regular copies of the program you are working on by simply entering GOTO 2. One other small point which will save time in the future is that a module such as this one, attached to the front of all your programs, gives a standard start line for the program—line one. It is very often desirable to start a program

with GOTO if variables have been set up that you do not wish to clear with RUN—with this module in place, you do not have to determine the number of the first line, you know that you can always start with GOTO 1.

The module preceded all the Basic programs in the book as they were developed but, since only the program name changes, it will not be included in the rest of the programs listed in the book.

2.4.1.1. *Lines 1-4*

```
1 GOTO 4
2 SAVE "banker.",a
3 STOP
4 REM
```

2.4.2. Initialisation

Every program worth the name uses variables and constants, that is to say, labels whose values can be changed during the course of the program. In general, it is good practice to declare the value of major variables at the very beginning of the program and this process is known as initialisation.

In the case of this particular program, the object of the module is to set up the arrays which will be used to store the eventual data, together with some useful string functions which later on will save a great deal of time in programming.

2.4.2.1. *Lines 10000-10170*

```
10000 ' *****
10010 ' Initialisation
10020 ' *****
10030 cls$=CHR$(27)+"E"+CHR$(27)+"H"
10040 rvson$=CHR$(27)+"p"
10050 rvsoff$=CHR$(27)+"q"
10060 DEF FN locate$(x,y)=CHR$(27)+"Y"+CHR$(32+y)+
      CHR$(32+x)
10070 WHILE in=0
10080   in=1
10090   DIM a$(1000,1),a(1000,1):a(0,1)=999
10100   RESTORE
10110   DIM mo$(11)
10120   FOR i=0 TO 11
10130     READ mo$(i)
```

```
10140 NEXT i
10150 DATA January,February,March,April,May,June
10160 DATA July,August,September,October,
      November,December
10170 WEND
```

2.4.2.2. *Commentary*

Lines 10030-10050: These three strings contain 'control characters' which can be used to give instructions to the system. The particular strings defined here, whenever they are included in a PRINT statement, will clear the screen, or switch on or off reverse video. When these features are used in the program, there will be no need to spell out the string in full, only to use the name.

Line 10060: This line creates a user defined function, a series of steps to be carried out whenever the function name, in this case LOCATE\$, is mentioned. The particular steps in this function create a string of control characters which, when they are printed, position the cursor. Where the position will be depends upon the values specified when the function is used. PRINT FN LOCATE\$(10,20), for instance, would place the cursor at character position 10 across and 20 down.

Line 10070: This loop represents a very simple but useful technique called auto-initialisation. If at any time you stop the program and wish to restart it again without losing the data, you probably already know that the way to do it is to use the GOTO command, since RUN clears all the variables out of the memory. A command like GOTO 1 will restart the program without altering memory contents, but that is hardly of any use if the first module then re-dimensions all the arrays. What this loop does is to detect the value of a variable called IN, which you will find in all the programs that follow. If the value of IN is not zero, then the program assumes correctly that there is data in memory and so does not redeclare the arrays. If IN is zero then the memory must have been cleared and the arrays are set up, with IN being set to -1 to record the fact. If you wish to restart the program and clear the arrays, then the RUN command will clear out the memory, including IN.

Line 10090: The array A\$ will be used to store the names of payments and a special string, explained later, which records the months in which the particular payment is made. The numerical array A will store the amount of each payment and the day of the month on which it is made. The setting of A(0,1) to 999, an impossible day of the month, is used by the later sorting routine to detect the end of the file.

Lines 10100-10160: This loop reads the names of the months of the year into the array MOS

2.4.3. The Program Menu

This module represents a technique which will play a large part in the programs within this book—the menu. The program presents a variety of possibilities to the user and it must be the user who, to a large extent, dictates what happens. This is done by presenting the user with a list of the choices the program provides and allows them to specify which is to be acted upon. More complex programs later in the book will make use of several menus, each reflecting the variety of choices under one main heading.

```

      BANKER
      =====

      Commands Available:

          1) New Payments
          2) Examine/Delete Payments
          3) Print Statement
          4) Save File
          5) Load File
          6) End

      Which do you require: █

```

Drive is A:

Fig. 2-2: The Banker program menu

2.4.3.1. Lines 11000-11230

```

11000 ' *****
11010 ' Menu
11020 ' *****
11030 z=0 : WHILE z<>6

```



```
11040 PRINT cls$ : PRINT TAB(30) "BANKER" :  
      PRINT TAB(30) "=====  
11050 PRINT : PRINT  
11060 PRINT TAB(20) "Commands Available:" :  
      PRINT  
11070 PRINT TAB(26) "1) New Payments"  
11080 PRINT TAB(26) "2) Examine/Delete Payments"  
11090 PRINT TAB(26) "3) Print Statement"  
11100 PRINT TAB(26) "4) Save File"  
11110 PRINT TAB(26) "5) Load File"  
11120 PRINT TAB(26) "6) End"  
11130 PRINT : PRINT TAB(20) : INPUT "Which do  
      you require: ",z  
11140 WHILE pa=0 AND (z=2 OR z=3 OR z=4)  
11150   PRINT : PRINT : PRINT TAB(16) "***** NO  
      DATA ENTERED YET *****" CHR$(7)  
11160   FOR i=0 TO 10000 : NEXT i  
11170   z=0  
11180 WEND  
11190 ON z GOSUB 12000,13000,14000,15000,16000  
11200 WEND  
11210 PRINT cls$  
11220 PRINT FN locate$(20,11) "CLOSED FOR  
      BUSINESS"  
11230 END
```

2.4.4. Entering new items

This input module is of greater complexity than many in this book, for the simple reason that the entries themselves are complex. For each item recorded, five facts need to be known: whether the payment is a credit or a debit (money received or money paid out), the name of the payment, the amount, the months in which the payment is due and the day of the month on which the payment is made.

2.4.4.1. Lines 12000-12460

```
12000 ' *****  
12010 ' Enter New Items  
12020 ' *****  
12030 PRINT cls$  
12040 PRINT "New items:" : PRINT  
12050 PRINT "1) Credit" : PRINT "2) Debit"
```

```

12060 PRINT : INPUT "Which do you require:",cd :
      cd=cd-1
12070 q$=""
12080 WHILE q$="" OR LEN(q$)>18
12090   PRINT : PRINT "Name of payment (18 chars
      max.)"; : INPUT ":",q$
12100 WEND
12110 PRINT : INPUT "Amount:",q
12120 en=1
12130 WHILE en
12140   PRINT : INPUT "Months (e.g.01040710):",r$ :
      PRINT
12150   en=0 : FOR i=1 TO LEN(r$) STEP 2
12160     m=VAL(MID$(r$,i,2))-1
12170     WHILE (m<0 OR m>11) AND en=0
12180       PRINT : PRINT "      ***** INVALID MONTH
      INPUT *****" CHR$(7)
12190       en=1 : i=LEN(r$) : FOR j=1 TO 3000 :
      NEXT j
12200   WEND
12210   IF en=0 THEN PRINT mo$(m) "/";
12220   NEXT i : PRINT : PRINT
12230 WEND
12240 INPUT "Day of payment:",s
12250 PRINT : INPUT "Are these correct (y/n)";t$
12260 IF LOWER$(t$)<>"y" THEN RETURN
12270 pa=pa+1
12280 j=pa-1
12290 WHILE s<a(ABS(j),1) AND j>=0
12300   FOR k=0 TO 1
12310     a$(j+1,k)=a$(j,k)
12320     a(j+1,k)=a(j,k)
12330   NEXT k
12340   j=j-1
12350 WEND
12360 j=j+1
12370 a$(j,1)=STRING$(12,"0")
12380 FOR i=1 TO LEN(r$) STEP 2
12390   m=VAL(MID$(r$,i,2))
12400   a$(j,1)=LEFT$(a$(j,1),m-1)+"1"+RIGHT$(a$(j,
      1),12-m)
12410 NEXT i
12420 a$(j,0)=q$

```

```
12430 a(j,0)=q
12440 a(j,1)=s
12450 IF cd=1 THEN a(j,0)=-a(j,0)
12460 RETURN
```

2.4.4.2. *Commentary*

Line 12060: The program clearly needs to know whether the item is to be paid out or received, debit or credit. This is recorded in the form of the variable CD (Credit/Debit).

Lines 12130-12230: The months in which the payment is to be made are input in the form of a string of two digit numbers, with no separation between them. Thus if a quarterly payment were to be needed in February, May, August and November, the input would be '02050811', representing months 2,5,8 and 11. The FOR loop beginning at line 12150 scans the string input to make sure that it does yield a series of sensible month values and informs the user if an error has been made. As a further check, the loop prints out the names of the specified months as recorded in MO\$, so that the user can determine that they are not only sensible but are the months intended. The routine is repeated if an invalid month input has been made.

Line 12270: At this point, the information input has been checked and confirmed by the user, so the variable recording the number of items in the file, PA, is incremented by one.

Lines 12290-12360: The purpose of this loop is to place the new item into correct day order within the file. Rather than make duplicate copies of payments which fall in more than one month, the method adopted is to store all of the entries once only, in day order. When the statement for a particular month is requested, a later part of the program will run through all of the entries, checking each to see if it falls on or before the specified month and so need be taken into account when determining the balance. When inserting the new item, the loop from 12290 to 12350 starts with the highest day value (which is the dummy 999 inserted in the initialisation module) and works its way down until it finds a payment with a day value *less than* that of the item just input by the user. If it does *not* find the correct position, it moves the item it has just examined up one place. In other words, it scans down the file, moving a spare line with it. When the correct position is found, the spare line is already in the right place. As a final action the value of J, which records the position of the first entry found which has a day of payment less than S, is increased by one to point to the spare line.

Lines 12370-12410: The next task is to translate the list of months input into a format which can easily be scanned by later parts of the program. The simple expedient adopted is to use a string of 12 zeroes, recording months in which the payment is not to be made, then to use a loop to change a zero to one for any months in which payment is to be made. Thus, in the case of our quarterly example above, the eventual string would read '010010010010'.

Lines 12420-12450: The new information is placed into the main arrays. If the variable CD records that the item is a debit, the amount is multiplied by minus one, making it negative.

2.4.5. Displaying the statement

Though there are more modules to come, the final task for the main part of the program is to take the items which have been entered using the previous module and compile them into a statement for any specified month on the year. The statement will include a calculation of the balance carried forward from previous months and will also display in full all the payments for the month, and the continuing balance created by each payment.

2.4.5.1. Lines 14000-14480

```

14000 ' *****
14010 ' Compile Statement
14020 ' *****
14030 sum=0
14040 PRINT cls$ : PRINT "Statement:" : PRINT
14050 q=0 : WHILE q<1 OR q>12
14060 PRINT : INPUT "Number of month for
      statement (1-12):",q
14070 WEND
14080 FOR j=1 TO q-1
14090 FOR i=0 TO pa-1
14100 IF MID$(a$(i,1),j,1)="1" THEN sum=sum+a(i,
      0)
14110 NEXT i
14120 NEXT j
14130 PRINT : INPUT "Send statement to printer (y/
      n)";p$
14140 IF LOWER$(p$)="y" THEN prn=-1 ELSE prn=0
14150 OPEN "o",1,"$$$$$$$$.$$$"
14160 PRINT #1,"Statement:" : PRINT #1,

```



```
14170 PRINT #1,TAB(20-LEN(mo$(q-1))/2) UPPER$(mo$(
q-1))
14180 PRINT #1,STRING$(39,"-")
14190 PRINT #1,"DAY & DETAILS";TAB(26);"ITEM
BALANCE"
14200 PRINT #1,STRING$(39,"-")
14210 PRINT #1,"    Balance forward" TAB(32) USING
"####.##-";sum
14220 FOR i=0 TO pa-1
14230   d=0 : WHILE MID$(a$(i,1),q,1)="1" AND d=0 :
      d=1
14240   PRINT #1,USING "## &";a(i,1);a(i,0);
14250   PRINT #1,TAB (23);USING "####.##-";a(i,0);
14260   sum=sum+a(i,0)
14270   PRINT #1,TAB (32);USING "####.##-";sum
14280   WEND
14290 NEXT i
14300 CLOSE 1
14310 count=0
14320 OPEN "i",1,"#####.###"
14330 IF NOT prn THEN PRINT cls$
14340 WHILE NOT EOF(1)
14350   count=count+1
14360   LINE INPUT #1,t$
14370   IF prn THEN LPRINT t$ ELSE PRINT t$
14380   quit = prn
14390   WHILE (count>20 OR EOF(1)) AND NOT quit
14400     quit=-1
14410     count=0
14420     PRINT : PRINT "Press a key ..." : WHILE
INKEY$="" : WEND
14430     PRINT cls$
14440   WEND
14450 WEND
14460 CLOSE 1
14470 KILL "#####.###"
14480 RETURN
```

2.4.5.2. *Commentary*

Line 14030: The variable SUM will be used to hold the balance in the account—both the balance carried forward and the balance after each item.

Lines 14080-14120: Provided that the statement is not for the first month, in which case there is no balance to be carried forward, these two loops scan the whole of the payments list once for each month which precedes the month of the statement. In this way, each payment is examined to see whether it is made in each of the preceding months, in which case the appropriate amount is added to the total in SUM. By the end of the two loops, SUM contains the full total of any changes in the balance since the beginning of the year. (Keeping a complete balance, including any monies which were in the account at the beginning of the year, can be easily achieved by entering the balance from the end of the previous year as a payment on January the first.)

Line 14140: The variable PRN is used to store the user's decision as to whether the output is to be sent to a printer, with 'yes' represented by 1 and 'no' by 0.

Line 14150: The module will output the finished copy of the account to either the screen or the printer. To simplify this task, what happens is that the account is created in a temporary file on the disc and then, when it is completed, read back to one or the other destination. Doing it in this manner will allow us, at a later point, to make one decision about the destination for the data rather than the whole module being filled with IFs every time something is printed.

Lines 14220-14290: This loop scans through the complete list of payments, while the WHILE loop at 14230 selects only those which have a '1' in the relevant position of the string recording the months in which the payment is to be made. When a payment is to be made in the month specified for the statement, the loop prints out the day, A(I,1), the name, A\$(I,0), the amount, A(I,0), and finally the balance the payment produces, obtained by adding the amount to the previous total in SUM. The screen is maintained in orderly columns, despite the fact that figures may vary in length, by the use of TAB, which starts items at a standard position on the screen, and PRINT USING, which imposes a standard format on the figures displayed.

Lines 14320-14460: As we saw earlier in the module, the statement is not sent initially to the screen or printer, but to a temporary file on the disc. What these lines do is to read each line from the disc file and then send it to either destination, according to the decision previously made by the user. The loop at line 14420 is simply a delaying tactic. If the output is being sent to the screen, then every 20 lines (and at the end of the statement) the output will pause until a key is pressed. Without some such provision, the statement would simply scroll quickly up the screen and return to the main menu before anyone had time to read it.

2.4.6. Data Files

We now turn for the first time to an area that many people neglect to their cost in their own programs—the storage of data on disc. As you have no doubt found if you have made one or two mistakes in the entry of a program, re-entering the data to test it further can become more than a little irritating after a while. In addition, for many programs, there is little point in putting the data items into the computer if you have to remember them anyway every time you switch the computer off.

All these problems can be overcome by the use of the disc drive to permanently record the data which has been entered and, having recorded it, to reload it into the memory whenever you wish.

2.4.6.1. Lines 15000-16140

```
15000 ' *****
15010 ' Save file
15020 ' *****
15030 PRINT cls$ : q$="" : WHILE LOWER$(q$)<>"y"
15040 INPUT "Name of file to be saved: ";fi$
15050 PRINT : PRINT "File to be saved is " fi$
15060 PRINT : INPUT "Is this correct (y/n)";q$
15070 WEND
15080 OPEN "o",1,fi$
15090 WRITE #1,pa
15100 FOR i=0 TO pa
15110 WRITE #1,a$(i,0),a$(i,1),a(i,0),a(i,1)
15120 NEXT i
15130 CLOSE 1
15140 RETURN
16000 ' *****
16010 ' Load file
16020 ' *****
16030 PRINT cls$ : q$="" : WHILE LOWER$(q$)<>"y"
16040 INPUT "Name of file to be loaded: ";fi$
16050 PRINT : PRINT "File to be loaded is ";fi$
16060 PRINT : INPUT "Is this correct (y/n)";q$
16070 WEND
16080 OPEN "i",1,fi$
16090 INPUT #1,pa
16100 FOR i=0 TO pa
16110 INPUT #1,a$(i,0),a$(i,1),a(i,0),a(i,1)
```



```
16120 NEXT i
16130 CLOSE 1
16140 RETURN
```

2.4.6.2. *Commentary*

Lines 15030-15070: These lines allow the user to specify the name of the file in which the data is to be stored. A parallel set of lines in the second module allow the name of the file to be loaded to be specified. In this way the program can work on several sets of data in succession, simply by changing the name of the file to be used.

Line 15080: Before data can be stored on the disc, a place must be prepared for it, a process known as 'opening a file'. In this line the computer is told that the file is going to be used to hold output from the program, that the file will henceforward be referred to by the program as file number one and the name to be given to the file on the disc.

Lines 15090-15120: Having OPENed the file, all that we need to do is to print information to it. This is done with the command WRITE. The items printed to the file consist of the contents of the main arrays plus the variable PA.

Line 15130: Finally, the file must be CLOSED. Failure to do this will mean that the file number will not be available for future use and it may even lead to the data on the disc being lost.

Lines 16000-16140: These lines perform the opposite function to those above, in that they recall previously stored data from the disc.

Line 16080: Once again, a file must be OPENed. The only difference between this and the previous OPEN statement is that 'i' tells the system that the file is to be used to input data to memory.

2.4.7. Examining and deleting items

When the program comes into actual use, the majority of the data which it is used to store will never be displayed in detail. When monthly statements are compiled there will be no way for the user to know whether a particular payment shown for that month is to be made in other months, for instance. To allow the checking of the full details associated with a particular payment, another module is needed which is dedicated to that purpose.

The current module is designed to allow the user to page through the entries in the main arrays examining the details and, if necessary, deleting entries.

2.4.7.1 Lines 13000-13250

```
13000 ' *****
13010 ' Examine/Delete Items
13020 ' *****
13030 FOR i=0 TO pa-1
13040   PRINT cls$
13050   PRINT "Payment:" a$(i,0)
13060   PRINT "Amount: " a(i,0)
13070   PRINT "Months: ";
13080   FOR j=1 TO 12
13090     IF MID$(a$(i,1),j,1)="1" THEN PRINT mo$(j-
13100       1) "/";
13110   NEXT j : PRINT
13120   PRINT "Day of payment:" a(i,1)
13130   PRINT : PRINT "Commands available:" :
13140   PRINT rvson$ "RETURN" rvsoff$ " next item"
13150   PRINT "'[ quit"
13160   PRINT "'[D delete item"
13170   q$="" : PRINT : INPUT "Which do you
13180     require:",q$
13190   WHILE UPPER$(q$)="[D"
13200     FOR j=i TO pa-1
13210       FOR k=0 TO 1
13220         a$(j,k)=a$(j+1,k) : a(j,k)=a(j+1,k)
13230       NEXT k,j
13240       pa=pa-1 : q$="[ "
13250     WEND
13260   IF q$="" THEN NEXT i
13270 RETURN
```

2.4.7.2. Commentary

Line 13030: This loop will allow the user to scan through the items in the file one by one. This is a very simple method but later programs will show how easy it is to give the user the ability to move backwards and forwards through the file.

Lines 13050-13110: The full details of the entry, including all the months in which it is paid, are displayed.

Lines 13120-13160: The user is presented with a three option menu allowing return to the main program menu, the display of the next entry in the file or the deletion of the item currently displayed.

Lines 13170-13230: If the user specifies deletion, the item is removed from the file by the simple expedient of copying all the entries later in the file down one position, thus overwriting the entry to be deleted. The nested loops are necessary since there are two items on every line of the arrays A and A\$ and both must be copied down one line.

2.5. Accountant

The second program in this chapter is more complex than Banker. Its function is to keep two sides of a simple set of accounts, setting them out in the traditional format, with some items standing alone and others clearly divided into groups representing different types of expenditure. The accounts generated using the program can be displayed on the screen or output to a printer if desired.

	DEBIT	
Purchase car	500.00	
Spares		
Petrol tank	37.95	
Door	50.00	

		87.95
Running cost		
Petrol	20.53	

		20.53

TOTAL:	608.48	
Press a key . . .		

Drive is A:

Fig. 2-3: A typical account prepared by Accountant

2.5.1. Initialisation

A standard initialisation module.

2.5.1.1. Lines 10000-10070

```
10000 ' *****
10010 ' Initialise
10020 ' *****
10021 cls$ = CHR$(27)+"E"+CHR$(27)+"H"
10022 rvson$ = CHR$(27)+"p"
10023 rvsoff$ = CHR$(27)+"q"
10024 DEF FN locate$(x,y) = CHR$(27)+"Y"+CHR$(32+y)+CHR$(32+x)
10030 PRINT cls$
10040 WHILE in=0
10050   in=1
10060   DIM a$(1,99),a(1,99)
10070 WEND
```

2.5.1.2. Commentary

Line 10060: The two sides of the accounts, credit and debit, including the names associated with each payment, are stored in the two sides of the arrays A and A\$. Up to 100 items can be stored on both sides, though you can increase that number if you wish to alter the DIM statements given here and you have memory available.

2.5.2. The main menu

A standard menu module with protection against accessing certain functions before any data has been entered.

ACCOUNTANT

=====

Commands Available:

- 1) New Headings
- 2) Change/Delete Items
- 3) Print Accounts
- 4) Save File
- 5) Load File
- 6) End

Which do you require: █

Drive is A:

Fig. 2-4: The Accountant main menu

2.5.2.1. Lines 11000-11220

```

11000 ' *****
11010 ' Menu
11020 ' *****
11025 z=0 : WHILE z<>6 : PRINT cls$
11030 PRINT TAB(30) "ACCOUNTANT" : PRINT TAB(30)
      "=====
11050 PRINT FN locate$(20,9) "Commands Available:
11060 PRINT FN locate$(25,11) "1) New Headings"
11070 PRINT TAB(26) "2) Change/Delete Items"
11080 PRINT TAB(26) "3) Print Accounts"
11090 PRINT TAB(26) "4) Save File"
11100 PRINT TAB(26) "5) Load File"
11110 PRINT TAB(26) "6) End"
11120 PRINT FN locate$(20,20) ; : LINE INPUT "
      Which do you require: ",z$
11130 IF z$<"4" AND z$>"0" THEN GOSUB 12000
11135 z = VAL(z$)
11140 WHILE c(cd)=0 AND (z=2 OR z=3 OR z=4)

```



```
11150 PRINT : PRINT : PRINT CHR$(7) " *****
      NO DATA ENTERED YET *****" : FOR i=1 TO 1500
      : NEXT i
11160 z=0
11170 WEND
11180 ON z GOSUB 13000,16000,18000,19000,20000
11190 WEND
11200 PRINT cls$
11205 PRINT FN locate$(30,15) rvson$ SPC(24)
      rvsoff$
11210 PRINT FN locate$(30,16) rvson$ " PROGRAM
      TERMINATED " rvsoff$
11215 PRINT FN locate$(30,17) rvson$ SPC(24)
      rvsoff$
11217 PRINT : PRINT
11220 END
```

2.5.3. Credit or Debit?

Unlike Banker, several parts of this program need to know whether a credit or debit item is being specified, so the routine to request this information is included in a separate module.

2.5.3.1. Lines 12000-12110

```
12000 ' *****
12010 ' Credit or Debit?
12020 ' *****
12030 cd=-1 : WHILE cd<>0 AND cd<>1
12040 PRINT cls$
12050 PRINT "Credit or debit:" : PRINT
12060 PRINT "1) Credit" : PRINT "2) Debit"
12070 PRINT : INPUT "Which do you require:",cd :
      cd=cd-1
12090 IF cd=1 THEN cd$="DEBIT" ELSE cd$ = "
      CREDIT"
12100 WEND
12110 RETURN
```

2.5.4. The type of item

This module is trivial in itself but it gives a clue as to why the program is bound to be longer than something like Banker. The purpose of the module is to allow the user to specify which of three types an item about to be input falls under.

The three types are:

- 1) A single item: All that is required for this is the name of the item and the amount.
- 2) A main heading: It is this type which allows groups of items to be specified within the overall account. If you were using the program to prepare domestic accounts, for instance, you might set up 'CAR' as a main heading for a group of items including items like tyres, fuel, repairs and so on.
- 3) Sub-headings: As illustrated under 2) above, each main heading can have a list of items following it, which are part of a separate group.

2.5.4.1. Lines 13000-13120

```

13000 ' *****
13010 ' Input Headings
13020 ' *****
13030 PRINT cls$
13040 PRINT TAB(30) "New items:" : PRINT
13050 PRINT TAB(32) cd$
13060 PRINT FN locate$(19,9) "Is the item:" :
      PRINT
13070 PRINT TAB(26) "1) A Single Item;"
13080 PRINT TAB(26) "2) A Main Heading or"
13090 PRINT TAB(26) "3) A Sub-Heading"
13100 PRINT : PRINT TAB(20) ; : INPUT "Enter type
      required (or 0 to quit)";ty
13110 ON ty GOSUB 14000,14000,15000
13120 RETURN

```

2.5.5. Entry of single items and main headings

Two separate modules take care of the input of sub-headings on the one hand, and single items or main headings on the other. It is important, in understanding later parts of the program, that you try to follow the way in which the items are stored and the special indicator characters which record the item type.

2.5.5.1. Lines 14000-14120

```

14000 ' *****
14010 ' Single Item or Main Heading
14020 ' *****

```

```
14030 q=0 : q$="" : r$=""
14040 PRINT : INPUT "Name of item:",q$
14050 IF ty=1 THEN PRINT : INPUT "Amount for item:
      ",q
14060 PRINT : INPUT "Is this correct (y/n)";r$
14070 IF LOWER$(r$)<>"y" THEN PRINT : PRINT CHR$(
      7) "***** NOT REGISTERED *****" : FOR
      i=1 TO 1500 : NEXT i : RETURN
14080 IF ty=1 THEN q$="%" + q$ ELSE q$="*" + q$
14090 a$(cd,c(cd))=q$
14100 a(cd,c(cd))=q
14110 c(cd)=c(cd)+1
14120 RETURN
```

2.5.5.2. *Commentary*

Line 14050: As mentioned in the introduction to the previous module, main headings have no money figure associated with them, so this line accepts a figure only for single items.

Line 14080: There are no separate storage areas for the different types of item, apart from the credit and debit sides of the arrays. Later parts of the program will determine the item type by looking at a special indicator character attached to the beginning of the item name. This will be '%' for a single item and '*' for a main heading.

Lines 14090-14120: You have already met the variable CD, which records whether an item is a credit or a debit. Here CD is used to decide which side of the arrays A and A\$ the new item is to be placed. In addition, we need to keep a record of the number of items on the credit and debit sides, since these will normally be different. This is done by the array C. The array did not need to be declared in the initialisation module since it will have only two elements, C(0) and C(1), corresponding to the credit and debit sides of the main arrays. Once again, the value of CD is used to indicate which of the two elements of C is to be used. Applying this, we can see that when reference is made to:

A(CD,C(CD))

what is meant is:

- 1) An element in the numeric array A.
- 2) On the side indicated by the value of CD, ie, credit or debit.

3) The first empty element on that side, determined by the record kept in C of how many items are already stored on that side.

2.5.6. Entering a sub-heading

The question of entering a new sub-heading is not quite as simple as that for a single item. For each new sub-heading that is entered, a check has to be made for the presence of the relevant main heading and the item placed next to its main heading rather than simply tagged on to the end of the items previously stored.

2.5.6.1. Lines 15000-15210

```

15000 ' *****
15010 ' Sub Heading
15020 ' *****
15030 PRINT : INPUT "Main heading:",q$
15040 q$=" "+q$
15050 p1=-1 : FOR i=0 TO c(cd)-1
15060   IF a$(cd,i)=q$ THEN p1=i+1
15070 NEXT i
15080 IF p1=-1 THEN PRINT : PRINT CHR$(7) "*****
      NO HEADING OF THAT NAME *****" : FOR i=1 TO
      1500 : NEXT i : RETURN
15090 PRINT : INPUT "Name of sub-heading:",q$
15100 PRINT : INPUT "Amount:",q
15110 PRINT : INPUT "Are these correct (y/n)";r$
15120 IF LOWER$(r$)<>"y" THEN PRINT : PRINT CHR$(
      7) "***** NOT REGISTERED *****" : FOR
      i=1 TO 1500 : NEXT i : RETURN
15130 q$=" "+q$
15140 FOR i=c(cd)+1 TO p1+1 STEP -1
15150   a$(cd,i)=a$(cd,i-1)
15160   a(cd,i)=a(cd,i-1)
15170 NEXT i
15180 a$(cd,p1)=q$
15190 a(cd,p1)=q
15200 c(cd)=c(cd)+1
15210 RETURN

```


2.5.6.2. Commentary

Lines 15030-15080: The name of the relevant main heading is requested and a check is made of the items already in the file to ensure that the heading actually exists.

Lines 15140-15200: As previously mentioned, the whole point of a sub-heading is that it should appear in the main accounts as part of a group printed under the relevant main heading. In order to achieve this simply, the method employed is to store it in the file next to its main heading. The position of the first item following the main heading has already been found by the FOR loop at 15050, so all that needs to be done is to move up all the items from that point and place the new item into the array at the correct point.

2.5.7. Data files

Two standard data file modules of the kind described during the commentary on the last program.

2.5.7.1. Lines 19000-20160

```
19000 ' *****
19010 ' Save
19020 ' *****
19030 PRINT cls$ : q$="" : WHILE LOWER$(q$)<>"y"
19040 INPUT "Name of file to be saved:";fi$
19050 PRINT : PRINT "File to be saved is " fi$
19060 PRINT : INPUT "Is this correct (y/n)";q$
19070 WEND
19080 OPEN "o,",1,fi$
19090 FOR i=0 TO 1
19100 WRITE #1,c(i)
19110 FOR j=0 TO c(i)-1
19120 WRITE #1,a$(i,j),a(i,j)
19130 NEXT j
19140 NEXT i
19150 CLOSE 1
19160 RETURN

20000 ' *****
20010 ' Load
20020 ' *****
20030 PRINT cls$ : q$="" : WHILE LOWER$(q$)<>"y"
```

```

20040 INPUT "Name of file to be loaded:";fi$
20050 PRINT : PRINT "File to be loaded is ";fi$
20060 PRINT : INPUT "Is this correct (y/n)";q$
20070 WEND
20080 OPEN "i",1,fi$
20090 FOR i=0 TO 1
20100 INPUT #1,c(i)
20110 FOR j=0 TO c(i)-1
20120 INPUT #1,a$(i,j),a(i,j)
20130 NEXT j
20140 NEXT i
20150 CLOSE 1
20160 RETURN

```

2.5.8. Changes to items

A simple editing module with some added features to take account of the fact that some items do not stand alone but as part of groups of items under a common main heading.

2.5.8.1. Lines 16000-16290

```

16000 ' *****
16010 ' Changes and Deletions
16020 ' *****
16030 FOR i=0 TO c(cd)-1
16040 d=0 : WHILE d=0 : d=1
16050 PRINT cls$
16060 PRINT TAB(35) "Change or Delete:" : PRINT
16070 IF LEFT$(a$(cd,i),1)<>"$" THEN PRINT MID$(
a$(cd,i),2);
16080 IF LEFT$(a$(cd,i),1)="*" THEN hh$=MID$(a$(
cd,i),2) : PRINT
16090 IF LEFT$(a$(cd,i),1)="$" THEN PRINT hh$ :
PRINT " ";MID$(a$(cd,i),2);
16100 IF a(cd,i)<>0 THEN PRINT TAB(32) ; USING"#
###.##" ; a(cd,i)
16110 PRINT : PRINT TAB(26) "Commands available:
" : PRINT
16120 PRINT TAB(30) "1) Next Item"
16130 PRINT TAB(30) "2) Change Amount"
16140 PRINT TAB(30) "3) Return to Menu"
16150 PRINT TAB(30) "4) Delete Item"

```

```
16160 PRINT : PRINT TAB(26) "Which do you
      require ? ";
16170 LINE INPUT q$
16200 IF q$="4" THEN GOSUB 17000 : RETURN
16210 IF q$="3" THEN RETURN
16220 WHILE q$="2" AND LEFT$(a$(cd,i),1)<>"*"
16230 PRINT : INPUT "Amount to be added:",q
16240 PRINT : INPUT "Is that correct (y/n)";r$
16250 IF LOWER$(r$)="y" THEN a(cd,i)=a(cd,i)+q:
      q$=""
16260 WEND
16270 WEND
16280 NEXT i
16290 RETURN
```

2.5.8.2. *Commentary*

Lines 16070-16090: If the item recalled from the file is a single item, then it is printed—though stripped of the indicator character which is tagged on to the beginning of the name. If the item is a main heading, not only is it printed, but its name is stored in HH\$ so that it can be printed out above any of its sub-headings which follow.

Lines 16220-16260: Apart from deleting items, changes can be made to the value associated with a heading. This is done by entering a positive or negative figure by which the value of an item may be changed—not an absolute value which the item is to take. The advantage of this is that most changes which result in the need to add amounts to existing items as further expenditures or receipts, are made under items which already exist. Thus, if an extra 100 pounds is to be spent on car repairs, for which there is already a heading, all that needs to be done is to page through the file to that heading and enter '100'.

2.5.9. **Deleting items**

One final facility to be added in relation manipulating to existing items is deletion. In the case of Accountant, the deletion module is more complex than previous examples of the type. The reason for this is the existence of the groups formed around main headings. While there are no difficulties associated with the deletion of a single item or a sub-heading, it is clearly not such a simple matter when a main heading is deleted. In that case, not only has the main heading to be taken out, but all the sub-headings associated with it—otherwise the account would become clogged with sub-headings not attached to a main heading, making nonsense of the display.

2.5.9.1. Lines 17000-17140

```

17000 ' *****
17010 ' Deletions
17020 ' *****
17030 pl=i : gr=1
17040 d=0 : WHILE LEFT$(a$(cd,pl),1)="*" AND d=0 :
        d=1
17050 WHILE LEFT$(a$(cd,pl+gr),1)="$"
17060     gr=gr+1
17070 WEND
17080 WEND
17090 FOR k=pl TO c(cd)-gr-1
17100     a(cd,k)=a(cd,k+gr)
17110     a$(cd,k)=a$(cd,k+gr)
17120 NEXT k
17130 c(cd)=c(cd)-gr
17140 RETURN

```

2.5.9.2. Commentary

Line 17030: The position at which the deletion is to take place is sent from the previous module in the form of the variable I. This is transferred for the purposes of the current module to PL. The variable GR (GRoup) records how many items need to be deleted. It is initially set to one, and will only be increased if the item to be deleted is a main heading with sub-headings attached.

Lines 17040-17080: These two embedded loops will only be activated if the item specified for deletion is a main heading. The inner loop scans down the following entries, counting how many of those items which follow are preceded by a , indicating that they are sub-items for the main heading. The result of the count is kept in GR.

Lines 17090-17120: A typical loop to collapse an array and delete an item. The difference here is that instead of copying item X into space X-1 and therefore copying each element down one place, items are transferred GR places, thus deleting GR items, ie, all the sub-entries in the group based around the main item which is being deleted.

2.5.10. Displaying the accounts

After all the preparation, the one module which makes sense of the whole

thing, by displaying the account in its final form. Like the equivalent module in Banker, it looks complex, but having seen the display once you will have no difficulty understanding why everything is arranged as it is.

2.5.10.1. Lines 18000-18500

```
18000 ' *****
18010 ' Print Accounts
18020 ' *****
18030 tt=0 : ss=0
18040 PRINT cls$ : PRINT "Print Accounts:" :
PRINT
18050 PRINT : INPUT "Send accounts to printer (y/
n)";p$ : prn = LOWER$(p$)="y"
18060 OPEN "o",1,"$$$$$$. $$$"
18070 PRINT #1,TAB(20-LEN(cd$)/2) cd$ : PRINT #1
18080 FOR i=0 TO c(cd)-1
18090   tt=tt+a(cd,i)
18100   IF LEFT$(a$(cd,i),1)="*" THEN PRINT #1
18110   IF LEFT$(a$(cd,i),1)="$" THEN PRINT #1," "
18120   PRINT #1,MID$(a$(cd,i),2);
18130   d=0 : WHILE LEFT$(a$(cd,i),1)<>"*" AND d=0
: d=1
18140   PRINT #1,TAB(18);
18150   IF LEFT$(a$(cd,i),1) "%" THEN PRINT #1,
TAB(29);
18160   PRINT #1,USING "####.##";a(cd,i);
18170   IF LEFT$(a$(cd,i),1)="$" THEN ss=ss+a(cd,
i)
18180 WEND
18190 PRINT #1
18200 WHILE ss<>0 AND LEFT$(a$(cd,i+1),1)<>"$"
18210   PRINT #1,TAB(18);"-----"
18220   PRINT #1,TAB(29) USING "####.##";ss
18230   ss=0
18240 WEND
18250 NEXT i
18260 PRINT #1,TAB(29);"-----"
18270 PRINT #1,"TOTAL:" TAB(29) USING "####.##";
tt
18275 CLOSE 1
18300 count = 1
18310 OPEN "i",1,"$$$$$$. $$$"
```

```

18320 IF NOT prn THEN PRINT cls$
18330 WHILE NOT EOF(1)
18340   count = count+1
18350   LINE INPUT #1,t$
18360   IF prn THEN LPRINT t$ ELSE PRINT t$
18370   quit = prn
18380   WHILE (count>20 OR EOF(1)) AND NOT quit
18390     quit = -1
18400     count = 0
18410     PRINT
18420     PRINT "Press a key . . ."
18440     WHILE INKEY$="" : WEND
18450     PRINT cls$
18460   WEND
18470 WEND
18480 CLOSE 1
18490 KILL "#####.###"
18500 RETURN

```

2.5.10.2. *Commentary*

Line 18050-18060: As with Banker, the easiest way to allow output to be sent either to the screen or printer is first to place it into a disc file.

Line 18090: The variable TT will be used to store the running total for the account as it is printed.

Lines 18100-18120: These lines print the item name. For a main item one blank line is printed first to separate it from what has gone before, while for a sub-heading, the name is inset by two spaces.

Lines 18130-18180: These lines deal with the printing of the amounts for single items and sub-headings. Sub headings will be printed at the eighteenth position along the line, single items at position twenty-nine. If the item being dealt with, then the sub-total for the items in the current group is stored temporarily in SS.

Lines 18200-18240: This loop operates only when a group is being processed, as indicated by the fact that SS is not equal to zero, and the next item is not part of the group—ie, the group is complete. The effect of the loop is to print the total for the group in the main column of figures at position 29 along the line.

2.6. Unifile

Unifile is a gem of a program which has been developed over the years in a series of micro-computer books. Readers of previous books have written to say that they are using it in their businesses, to teach school children about the way micros handle information, to help with clubs and voluntary organisations or simply to keep track of their books and records at home. Entering the program will not only provide you with a very useful tool but with a whole series of useful techniques which can be used in a wide range of programs of your own. Later in this chapter you will find a second filing program which works according to a different method, allowing the storage of quantities of data larger than the memory of the PCW.

```
ENTRY 1 :  
NAME :BLOGGS J  
ADDRESS:BLOGGS CLOSE  
:BLOGGS VILLAGE  
:MR. BLOGGSTOWN  
PHONE:333-444-555  
  
COMMANDS AVAILABLE:  
ENTER for next item  
'A' to amend  
'C' to continue search  
'#' followed by number to move pointer  
'I' to quit function  
  
Enter Choice: █
```

Drive is A:

Fig. 2-5: Typical display in the Unifile search mode

2.6.1. Setting up the structure of the file

Many books aimed at the home micro owner offer inferior filing programs which are extremely inflexible in use. It is built into the program that every time the user stores something, it will be under the headings, Name, Address, Phone No., or some similar structure. The beauty of Unifile is that while it will

certainly allow you to use such a structure, it will also allow you create other files with very different structures—perhaps with one heading, perhaps with 10—without making any changes to the program itself. Unifile is what we like to call a chameleon program: one that adapts to a variety of different uses, reacting the user in different ways depending on the task that it is performing at the time.

The purpose of this module is to initialise some variables, but most importantly to allow the user to set up the original file in exactly as desired.

2.6.1.1. *Lines 20000-20170*

```

20000 ' *****
20010 ' Create File
20020 ' *****
20030 PRINT cls$
20040 PRINT TAB(32) "NEW FILE" : PRINT TAB(32) "=="
      "====="
20050 PRINT : INPUT "Are you sure (y/n)";r$ : IF
      LEFT$(UPPER$(r$),1)<>"Y" THEN RETURN
20060 array$(0)="" : item$(0)="" : ptr(0)=0
20070 ERASE array$,item$,ptr
20080 DIM array$(1000)
20090 PRINT:INPUT"How many items in each entry";x
20100 DIM item$(x-1),ptr(x-1)
20110 PRINT
20120 FOR i=0 TO x-1
20130   PRINT "Name of item #" i+1; : LINE INPUT ":
      ",item$(i)
20140   item$(i)=UPPER$(item$(i))
20150 NEXT i
20160 in=-1
20170 RETURN

```

2.6.1.2. *Commentary*

Line 20070-20080: The information to be held in Unifile will be stored in the array ARRAY\$. This is first erased and then redimensioned to provide for 1001 entries to the file. You can increase that number to 2000 or more if you wish. The only thing to bear in mind is that each element of a string array, before it is used, takes up three bytes of memory. If you dimension the array at 5000 items, you will be using up 15000 bytes of memory before any information is stored. It is simply a matter of judgement—if you think you

will need less than 1000 entries in a file, then you may as well leave the DIM statement as it is and save a little memory. Note also, that because we are using one element of an array to store each entry, the maximum length of a single entry, including its separator characters, is 255 characters—the maximum length of string that Basic can deal with.

Lines 20090-20150: Part of the secret of Unifile's flexibility. For each entry, which you can think of as a filing card if it helps you, you can define for yourself how many items will appear in the entry and what their names are.

For the more technically minded, while we shall be calling the whole filing card an *entry* and the individual headings *items* in computing jargon they would be known as *records* and *fields* respectively. The use of the variables defined here will be explained during the course of the commentary on the program.

2.6.2. Starting the program

The facility to create a new file is not used on every occasion when the program is run. On most occasions the user will want to reload data from disc. This start up module simply gives the user the opportunity to say which of the two options is desired.

2.6.2.1. Lines 10000-10230

```
10000 ' *****
10010 ' Start
10020 ' *****
10030 c1s$=CHR$(27)+"E"+CHR$(27)+"H"
10040 rvson$=CHR$(27)+"p"
10050 rvsoff$=CHR$(27)+"q"
10060 DEF FN locate$(x,y)=CHR$(27)+"Y"+CHR$(32+y)+
      CHR$(32+x)
10070 WHILE NOT in
10080 PRINT c1s$
10090 PRINT TAB(36) "UNIFILE"
10100 PRINT TAB(36) "======"
10110 PRINT
10120 PRINT
10130 PRINT
10140 PRINT TAB(20) "COMMANDS AVAILABLE:"
10150 PRINT
10160 PRINT TAB(26) "1) Create new file"
```

```

10170 PRINT TAB(26) "2) Load file"
10180 PRINT TAB(26) "3) End program"
10190 PRINT
10200 PRINT TAB(20) "Enter choice: ";
10210 INPUT z
10220 ON z GOSUB 20000,21000,22000
10230 WEND

```

2.6.3. Menu

A standard menu module.

```

                UNIFILE
                =====

COMMANDS AVAILABLE:

    1) Create new file
    2) Load file
    3) Enter information
    4) Search/Display/Change
    5) Save file
    6) Stop

Enter choice: ? █

```

Drive is A:

Fig. 2-6: The Unifile main menu

2.6.3.1. Lines 11000-11220

```

11000 ' *****
11010 ' Main Menu
11020 ' *****
11030 WHILE NOT done
11040 PRINT cls$
11050 PRINT TAB(36) "UNIFILE"

```

```
11060 PRINT TAB(36) "====="
11070 PRINT
11080 PRINT TAB(20) "COMMANDS AVAILABLE:"
11090 PRINT
11100 PRINT TAB(26) "1) Create new file"
11110 PRINT TAB(26) "2) Load file"
11120 PRINT TAB(26) "3) Enter information"
11130 PRINT TAB(26) "4) Search/Display/Change"
11140 PRINT TAB(26) "5) Save file"
11150 PRINT TAB(26) "6) Stop"
11160 PRINT
11170 PRINT TAB(20) "Enter choice: ":
11180 INPUT z
11190 ON z GOSUB 20000,21000,12000,17000,18000,
      22000
11200 WEND
11210 done=0
11220 END
```

2.6.4. Stopping the program

This small module displays the termination message for the program and sets the value of the variable DONE to inform the menu that that program is to stop.

2.6.4.1 Lines 22000-22060

```
22000 ' *****
22010 ' Stop
22020 ' *****
22030 done=-1 : in=-1
22040 PRINT cls$ FN locate$(20,13) ;
22050 PRINT "FILING SYSTEM CLOSED" : PRINT :
      PRINT
22060 RETURN
```

2.6.5. Saving data

A standard module.

2.6.5.1. Lines 18000-18080

```

18000 ' *****
18010 ' Save File
18020 ' *****
18030 PRINT cls$ : PRINT TAB(32) "SAVE FILE" :
      PRINT TAB(32) "=====
18040 PRINT : INPUT "Name under which to save: ",
      fi$
18050 PRINT : OPEN "o",1,fi$ : PRINT #1,it :
      PRINT #1,x
18060 FOR i=0 TO it-1 : WRITE #1,array$(i) : NEXT
18070 FOR i=0 TO x-1 : WRITE #1,item$(i) : NEXT i
18080 CLOSE 1 : RETURN

```

2.6.6. Loading data

A standard module.

2.6.6.1. Lines 21000-21160

```

21000 ' *****
21010 ' Load File
21020 ' *****
21030 PRINT cls$ : PRINT TAB(32) "LOAD FILE" :
      PRINT TAB(32) "=====
21040 PRINT : INPUT "Are you sure (y/n)";r$ : IF
      LEFT$(UPPER$(r$),1)<>"Y" THEN RETURN
21050 array$(0)=" " : item$(0)=" " : ptr(0)=0
21060 ERASE array$,item$,ptr
21070 DIM array$(1000)
21080 PRINT : INPUT "Name of file to load: ",fi$
21090 PRINT : OPEN "i",1,fi$ : INPUT #1,it,x :
      DIM item$(x-1),ptr(x-1)
21100 FOR i=0 TO it-1
21110   INPUT #1,array$(i)
21120 NEXT i
21130 FOR i=0 TO x-1
21140   INPUT #1,item$(i)
21150 NEXT i
21160 in=-1 : CLOSE 1 : RETURN

```


2.6.7. A better way of searching

In this module, and the two that follow, we take a look at how a new entry is added to the main file contained in ARRAY\$. The core of the method is contained here, however, because it is this module which allows Unifile to search rapidly through a large file of entries to find the correct place to insert a new one, or to conduct a fast search for the presence of a key entry in the file.

The method is known as 'binary searching' and it can be used to dramatically reduce the time for searching in any programs you write which hold long lists of ordered data. Consider the following example:

A file has been established containing 2000 names. The task is to insert a new name into the file, in the correct alphabetical order. If we cheat and look at the list of names we can determine that the new name 'YOUNGER' should actually go into the file at position 1731, though the computer has no way of knowing this in advance.

One thing we could do is to set the computer examining the names one by one from the beginning. It will begin with 'ADAMS' and note that 'YOUNGER' should come after, then go on to 'ADAMSON' and so on. Eventually, after examining 1732 names, the search will it upon a name like 'YOUNGMAN' which must come *after* 'YOUNGER', so the correct position has been found.

This is a reliable method, but how much better if the number of comparisons made could be cut down a little. Well, in the case of our file of 2000 names, the whole process can be accomplished by just 11 comparisons. Here's how it's done:

The computer begins the search by examining the name in position 1024 of the file, because 1024 is the greatest power of two (2^{10}) which will fit into the total number of entries (2000). The name at position 1024 is found to be alphabetically less than 'YOUNGER' and so the computer adds $1024/2$, or 512, or 2^9 to the original search position, arriving at 1536. Once again, the name at this position is alphabetically less than 'YOUNGER' so this time 256, or 2^8 is added to 1536, making 1792. Now something different happen because the name at position 1792 is later in the alphabetical order than 'YOUNGER' so instead of adding 128, or 2^7 , it is subtracted from the search position, giving 1664.

The search goes on, adding or subtracting decreasing powers of two to build a search pattern that looks like this:

COMPARISON NO.	POSITION	ACTION
1	1024	+ 512
2	1536	+ 256
3	1792	-128
4	1644	+ 64
5	1728	+ 32
6	1760	-16
7	1744	-8
8	1736	-4
9	1732	-2
10	1730	+ 1
11	1731	The correct entry

Try it yourself for different numbers of entries and different target positions in the order, you will find that it always works.

2.6.7.1. Lines 13000-13110

```

13000 ' *****
13010 ' Binary Search
13020 ' *****
13030 IF it=0 THEN ss=0 : RETURN
13040 po=INT(LOG(it)/LOG(2)) : ss=2^po-1
13050 FOR i=po-1 TO 0 STEP -1
13060   ss=ss+2^i*((array$(ss)>te$)-(array$(ss)<te$
    ))
13070   IF ss<0 THEN ss=0
13080   IF ss>it-1 THEN ss=it-1
13090 NEXT i
13100 IF array$(ss)<te$ THEN ss=ss+1
13110 RETURN

```

2.6.7.2. Commentary

Line 13030: If there are no entries in the file, the calculation would lead to an error. This line avoids that situation.

Line 13040: These two expressions find the greatest power of two which will fit into the number of items in the file and then set the search pointer (SS) equal to that number. The -1 in the second expression takes account of the fact that the array is numbered from zero, not one.

Lines 13050-13090: This loop conducts the search, using reducing powers of two. The main file is contained in ARRAY\$ and the new entry in TE\$. The value of the pointer is set by two logical conditions which indicate whether TE\$ is greater or less than the entry at ARRAY\$(SS).

Line 13100: In some cases the position arrived at will be one below the correct position—in this case SS is increased by one.

2.6.8. Inserting an item

This module inserts the new entry into the position indicated by the variable SS. This is done by moving everything from position SS onwards, up one place.

2.6.8.1. Lines 14000-14070

```
14000 ' *****
14010 ' Insert Entry
14020 ' *****
14030 IF it=0 THEN GOTO 14070
14040 FOR i=it TO ss+1 STEP -1
14050   array$(i)=array$(i-1)
14060 NEXT i
14070 array$(ss)=te$ : it=it+1 : RETURN
```

2.6.9. Making entries to the file

Having entered the modules which do the real work, we can now proceed to the one the user will have contact with when placing material into the filing system. The function of the module is to prompt the user to input the correct number of items, in the correct order, for each entry; to combine those items into a single string which will fit into one line of ARRAY\$ and then to call up the two previous modules to insert the entry into the main file.

2.6.9.1. Lines 12000-12230

```
12000 ' *****
12010 ' New Entries
12020 ' *****
12030 WHILE it<1000
12040   te$=""
12050   PRINT cls$
12060   PRINT TAB(32) "NEW ENTRIES"
12070   PRINT TAB(32) "====="
```

```

12080 PRINT
12090 PRINT "Entry number" it+1
12100 PRINT
12110 PRINT "Enter item specified or '[' to
      return to main menu"
12120 PRINT
12130 FOR i=0 TO x-1
12140   PRINT item$(i); : LINE INPUT ":",q$
12150   IF q$="[" THEN RETURN
12160   te$=te$+UPPER$(q$)+"["
12170 NEXT i
12180 PRINT : PRINT "Please wait for a moment
      ... "
12190 GOSUB 13000 : GOSUB 14000
12200 WEND
12210 PRINT cls$
12220 PRINT "*** SORRY, NO MORE ENTRIES POSSIBLE *
      **"
12230 FOR i=1 TO 2000 : NEXT i : RETURN

```

2.6.9.2. Commentary

Lines 12030 and 12220-12230: Entries are only accepted if there is room for them, otherwise an error message is generated. Note that if you want to change the maximum number of entries you will need to change this reference to 1000.

Lines 12130-12170: These are the lines which request the input of the individual items for the new entry. The name for each item is taken from the array ITEM\$, which was set up in the initialisation module. Each item is input under the name Q\$ and the entry of new items is terminated if Q\$ turns out to be '[' at any point.

If the input is not '[' then the item is added to TE\$, which will contain the whole entry, and a '[' character is added to the end of the item. The purpose of the '[' character has nothing whatsoever to do with its earlier use in terminating input of information, it is purely there as an indicator to later parts of the program as to where each item ends and the next item begins. Thus an entry such as

```

SMITH
JOHN
11 ANYSTREET
ANYTOWN

```


would be stored in ARRAY\$ as:

SMITH[JOHN[11 ANYSTREET[ANYTOWN[

creating what is known as a 'packed string'. The reason that '[' is chosen, is that it is unlikely that there should be any urgent reason why anyone should want to include it in an entry—if you do, then choose another separator character. Note that all items input are translated into upper case letters. Later you will see that the same happens when items are searched for in the file. This prevents inadvertently entered capital or lower case letters making it difficult to find an item in the file. If you must use upper and lower case letters together, then these provisions can be removed without harm to the operation of the program, though you must be aware that Basic regards any lower case letter as being later in the alphabet than any upper case letter, which can make the order of your file look rather odd.

2.6.10. Identify items in a single entry

Before we can go on to the modules which allow data to be retrieved from the file and manipulated, we need to enter this one, whose task is to search through an entry and record the position of each '[' character or, in effect, the end of each item in the entry. The values are stored in the array PTR, and will only relate to the current entry. When another entry is required, this module will have to be called again to analyse it.

2.6.10.1. Lines 19000-19080

```
19000 ' *****
19010 ' Analyse Record
19020 ' *****
19030 pp=0
19040 FOR i=0 TO x-1
19050 ptr(i)=INSTR(pp+1,array$(s1),"[")
19060 pp=ptr(i)
19070 NEXT i
19080 RETURN
```

2.6.10.2. Commentary

Line 19050: Apart from the binary search module, where the variable SS is used to indicate the entry being examined, the rest of the program makes use of the variable S1 to record the position in ARRAY\$ of the current entry. The effect of the line is to search for each occurrence of the '[' character, starting one character past the place where the last one was found.

2.6.11. Searching for items in the file

We can now proceed to the module which makes the program *useful* by allowing the data which has been stored to be retrieved. The current module will allow entries to be retrieved by one of four methods:

- 1) One by one in order from the current position.
- 2) By jumping forwards or backwards a specified number of items.
- 3) By entering a key item—the first item in the entry—for a fast search.
- 4) By searching for any occurrence of a combination of characters, wherever it is within an entry.

2.6.11.1. Lines 17000-17430

```

17000 ' *****
17010 ' Search
17020 ' *****
17030 s1=0 : PRINT cls$ : PRINT TAB(32) "SEARCH" :
      PRINT TAB(32) "=====" : PRINT
17040 IF it=0 THEN PRINT "*** NO DATA ENTERED
      YET ***" ELSE GOTO 17070
17050 FOR i=1 TO 2000 : NEXT i
17060 RETURN
17070 PRINT "COMMANDS AVAILABLE:"
17080 PRINT : PRINT "Input item for normal search"
17090 PRINT "Precede with '*' for initial search"
17100 PRINT rvson$ "ENTER" rvsoff$ " for first
      item on file"
17110 te$="" : PRINT : INPUT "Input search
      command: ",te$ : te$=UPPER$(te$)
17120 IF LEFT$(te$,1)="*" THEN te$=MID$(te$,2) :
      GOSUB 13000 : te$="" : s1=ss
17130 p$="C" : WHILE p$="C"
17140   IF te$="" THEN GOTO 17200
17150   ff=0 : FOR i=s1 TO it-1
17160     pp=INSTR(array$(i),te$)
17170     IF pp<>0 THEN ff=1 : s1=i : i=i-1
17180   NEXT i
17190   IF ff=0 THEN RETURN
17200   p$=""

```

```
17210 WHILE p$="" AND it>0
17220 IF s1>it-1 THEN s1=it-1
17230 IF s1<0 THEN s1=0
17240 GOSUB 19000
17250 PRINT cls$ : PRINT "ENTRY " s1+1 ":" :
PRINT : pp=0
17260 FOR i=0 TO x-1
17270 PRINT item$(i) ":" MID$(array$(s1),pp+1,
ptr(i)-pp-1)
17280 pp=ptr(i)
17290 NEXT i
17300 PRINT : PRINT "COMMANDS AVAILABLE:"
17310 PRINT : PRINT rvson$ "ENTER" rvsoff$ "
for next item"
17320 PRINT "'A' to amend"
17330 PRINT "'C' to continue search"
17340 PRINT "'#' followed by number to move
pointer"
17350 PRINT "'I' to quit function"
17360 PRINT : LINE INPUT "Enter Choice: ",p$
17370 p$=UPPER$(p$):IF p$="" OR p$="C" THEN s1=
s1+1
17380 IF p$="C" THEN GOTO 17420
17390 IF p$="A" THEN GOSUB 15000 : p$=""
17400 IF LEFT$(p$,1)="#" THEN s1=s1+VAL(MID$(p$,
2)) : p$=""
17410 WEND
17420 WEND
17430 RETURN
```

2.6.11.2. *Commentary*

Line 17030: S1, as mentioned in the commentary on the last module, is the pointer to the current entry, and starts at zero every time a search is begun.

Lines 17070-17110: This is the start-up menu for the search module. On each search you will see it only once, when you begin. Using this menu, the type of search to be made is specified—if you wish to change the search type you will need to quit the current search and start again with this menu.

The term 'Normal Search' indicates a search for a given combination of characters—the first entry returned will be the first one in the file which contains those characters, in any position.

'Initial Search' refers to a search for an entry which *begins with* the combination of characters specified by the user. Thus, inputting '*SMI' would find an entry beginning with the SMI, though not necessarily the first one. If the string specified is not present at the beginning of any entry, the entry returned will be the one occupying the place it would be inserted if input as a new entry.

You can extend either the normal search or the initial search over more than one item in an entry by including a '[' character in the string to be searched for. Using this technique on a file where the first item of each entry was a surname and the second item a first name, an initial search for 'SMITH[A' would turn up any SMITH with the initial 'A', though again, not necessarily the first.

On a normal search, if the specified string is not found in any of the entries in the file, program execution will return to the main program menu.

Line 17120: This line does all the work necessary for an 'initial search', by stripping off the leading asterisk and simply calling up the binary search module to find the correct position.

Lines 17130 and 17410: The main loop which will repeat a normal search if the user so requests at a later menu. Note that the initial search routine does not fall in this loop since there is no point in repeating an initial search—the result will always be the same entry.

Lines 17150-17180: By this point in the execution of the module, any input must be a string to searched for using the normal search. This is done by scanning entry using INSTR. If an item is found, the variable FF is set to one to indicate this. The position is recorded in S1 and then the loop variable, I, is set to its highest value so that the loop will terminate.

Lines 17210: This loop will continue as long as P\$, the input to the subsequent search menu, remains at a 'null string'. The reason that the value of IT has to be included in the condition for the loop is that within this loop there will be provision to delete items. If all the items are deleted we shall need to exit from the loop or an error will be generated.

Lines 17220-17230: Within the loop, the user has the option to move around the file by number—these lines check on subsequent passes through the loop that the pointer has not been moved outside the valid range for the current number of entries.

Lines 17260-17290: Having called up the previous module, these lines make use of the information about the position of the separator characters to print the items which make up the current entry, together with the item title. For each pass through the FOR loop, characters with a position between the value of the variable PP and a value contained in the array PTR are printed. When each batch of characters has been printed, PP is reset to the current value in the array PTR and the loop variable, I, picks up the next value in PTR.

Lines 17300-17360: The menu which appears once an entry is displayed. The user has the option to move on to the next entry, to call up the AMEND function (not yet entered), to search further for the previously specified string, to move through the file a specified number of entries, or to return to the main program menu.

Lines 17370-17380: These two lines relate to the loop starting at 17140 and 17220. If the input is a null string (ie, RETURN is pressed), then the loop at 17220 is repeated, printing out the next entry indicated by S1. If C is entered, the loop at 17140 executes the search again, starting at the next entry.

Line 17390: The AMEND function, which has not yet been entered.

Line 17400: Inputting a number preceded by the '#' sign allows the user to move forward or backwards through the file. Setting P\$ to a null string simply executes the loop at 17220 to print out the entry arrived at.

2.6.12. Deleting entries

The final touches to the program are added by the following two modules, which allow entries to be changed or deleted. The DELETE module is added first, since it is used whenever an entry is changed.

2.6.12.1. Lines 16000-16040

```
16000 ' *****
16010 ' Delete Item
16020 ' *****
16030 FOR j=s1 TO it-1 : array$(j)=array$(j+1) :
      NEXT j
16040 it=it-1 : RETURN
```

2.6.13. Changing entries

The program would be of limited use to us if we were not able to change existing data—this module fills that gap.

2.6.13.1. Lines 15000-15200

```

15000 ' *****
15010 ' Change Entry
15020 ' *****
15030 te$="" : pp=0
15040 FOR i=0 TO x-1
15050 PRINT cls$ : PRINT "Entry " s1+1 ":"
15060 PRINT : PRINT item$(i) ":" MID$(array$(s1),
      pp+1,ptr(i)-pp-1)
15070 PRINT : PRINT "COMMANDS AVAILABLE:"
15080 PRINT
15090 PRINT rvson$ "ENTER" rvsoff$ " leaves item
      unchanged"
15100 PRINT "Input new item to replace one shown"
15110 PRINT "'[D]' deletes whole entry"
15120 PRINT "'[ ' leaves whole entry unchanged"
15130 PRINT : LINE INPUT "Which do you require: "
      ,q$
15140 q$=UPPER$(q$) : IF q$="[D]" THEN GOSUB
      16000 : RETURN
15150 IF q$="[ " THEN RETURN
15160 IF q$(">)" THEN q$=q$+"["
15170 IF q$="" THEN q$=MID$(array$(s1),pp+1,ptr(
      i)-pp)
15180 pp=ptr(i) : te$=te$+q$
15190 NEXT i : GOSUB 16000 : GOSUB 13000
15200 s1=ss : GOSUB 14000 : RETURN

```

2.6.13.2. Commentary

Line 15040: This loop, while it looks very similar to the loop which prints the entries in search module, is different in that it prints only one item at a time.

Line 15140: Input of '[D]' when any item is displayed deletes the whole entry of which that item is a part—note that an individual *item* cannot simply be deleted since the number of items per entry is fixed.

Line 15150: Input of '[' in response to any item returns execution to the search module. Any changes made to previous items in the entry will be ignored and the entry will be unchanged.

Line 15160: If any input is made other than '[D' or '[' then it is interpreted as being a replacement for the item displayed. The '[' separator is added to the end of the item, as in the new items module.

Line 15170: Pressing RETURN, without an input, copies the item displayed without changes. If only one item is to be changed, simply press RETURN, for all the others. Note the one difference between the string expression here and the one used to print the item in line 15060. The '-1' at the end is dropped so that the '[' separator character is not stripped off.

Lines 15180-15190: The amended entry is built up in TE\$. When the entry is complete, the original entry is deleted from the file. The reason for this is that the changes made may have altered the correct position of the entry in the ordered file. Once deleted, the entry is sent to the binary search module and re-inserted, with its position in the file having been copied into the variable S1, so that the search module will know which item to display if the position has been changed.

2.7. NNumber

Not all filing is concerned with words. One of the things which microcomputers do best is store and manipulate figures. The current program, 'NNumber' (short for "Name and Number"), allows you to store the names of items, the units in which they are usually measured and an associated quantity. Now before you say that you can't see a use for such a program, consider the average shopkeeper or even domestic cook.

The shopkeeper has a mass of items which are called stock. All of the items which make up the stock have names, they come in different units (box, bottle, bag, etc) and all have a very important quantity associated with them—their price. In order, therefore, to make a microcomputer help with stock-taking or make out an invoice, it must remember these three facts about each item. In the home, the foods we eat each have a name, they come in different units (spoonsful, pounds, pinch, etc) and if we are interested in their effect on our weight, then they all have a quantity associated with them—calories.

These are just two examples (you can think of many more for yourself) of the importance of being able to record names, units and an associated quantity for a whole variety of items.

The purpose of NNumber is to allow you to create a dictionary of items—up to 200 of them—together with the units in which they are measured and the values associated with those units. Based on that dictionary, you will be able to construct lists of items which the program will display and total the quantity for you. NNumber is as easy to use in adding up the calories for a day's recipes as it is in providing a total price for a collection of goods.

Search:

```
Item number: 1
stock:Widgets
Units:Bag
Quantity per bag: 5
```

Commands available:

```
Input item to be searched for
ENTER for next item
'#' then number to move pointer
'd' to delete item
'l' to quit
```

Which do you require? █

Drive is A:

Fig. 2-7: Typical screen display in the NNumber search mode

2.7.1. Initialisation

A standard module, the only thing worthy of note being that the user is asked to specify the type of item the program will be dealing with. This will be a name such as 'Food item' or 'Stock item'. The phrase input will be used during the course of the program to prompt the user to input another item.

2.7.1.1. Lines 10000-10090

```
10000 ' *****
10010 ' Initialise
10020 ' *****
10030 cls$=CHR$(27)+"E"+CHR$(27)+"H"
```



```
10032 DEF FN locate$(x,y) = CHR$(27)+"Y"+CHR$(32+
    y)+CHR$(32+x)
10034 PRINT cls$
10040 DIM array$(1000,1),array(1000),te$(100,1),
    te(100) : cu=0 : it=0
10050 PRINT TAB(30) "NNUMBER" : PRINT TAB(30) "===
    ==="
10060 PRINT : PRINT : PRINT
10080 INPUT "Load from disc (y/n)";q$
10090 IF LOWER$(q$)="y" THEN GOSUB 22000 ELSE
    PRINT : INPUT "Overall name for items:",nn$
```

2.7.1.2. Commentary

Line 10040: The array ARRAY\$ is used to record the item name and unit name for each of the items in the dictionary. The associated quantity for each unit is stored in the equivalent element of the array ARRAY.

TE\$ and TE will serve a similar purpose to ARRAY\$ and ARRAY% but for the 'current list' which is extracted from the dictionary.

2.7.2. Menu

A standard menu module.

```
          NNUMBER
          =====

COMMANDS AVAILABLE:
1) Display current list
2) Input to current list
3) Start new current list
4) Delete from current list
5) Add to dictionary
6) Examine dictionary items
7) Save data
8) Stop

Enter choice:█
```

Fig. 2-8: The NNumber main menu

2.7.2.1. Lines 11000-11210

```

11000 ' *****
11010 ' Main Menu
11020 ' *****
11030 WHILE z<>8
11040   PRINT cls$
11042   PRINT TAB(30) "NNUMBER" : PRINT TAB(30) "=="
      "====="
11050   PRINT : PRINT : PRINT TAB (20) "COMMANDS
      AVAILABLE:"
11060   PRINT
11070   PRINT TAB(26) "1) Display current list"
11080   PRINT TAB(26) "2) Input to current list"
11090   PRINT TAB(26) "3) Start new current list"
11100   PRINT TAB(26) "4) Delete from current list"
11110   PRINT TAB(26) "5) Add to dictionary"
11120   PRINT TAB(26) "6) Examine dictionary items"
11130   PRINT TAB(26) "7) Save data"
11140   PRINT TAB(26) "8) Stop"
11150   PRINT : PRINT TAB(20) : INPUT "Enter
      choice:",z
11160   WHILE (z=1 OR z=2 OR z=4 OR z=6 OR z=7)
      AND it=0
11170     x$="NO DATA YET"
11172     GOSUB 23000
11174     z=0
11176   WEND
11178   ON z GOSUB 12000,13000,14000,20000,15000,
      18000,21000
11180 WEND
11190 PRINT cls$
11200 PRINT FN locate$(31,11) "PROGRAM TERMINATED"
11210 END

```

2.7.3. Data files

Once again, both standard modules.

2.7.3.1. Lines 21000-23070

```

21000 ' *****
21010 ' Save
21020 ' *****

```

```
21030 PRINT cls$
21040 PRINT "Save data:" : PRINT
21050 INPUT "Name of file:",file$
21060 OPEN "o",1,file$
21070 WRITE #1,nn$,cu,it
21100 FOR i=0 TO cu-1
21110 WRITE #1,te$(i,0),te$(i,1),te(i)
21140 NEXT i
21150 FOR i=0 TO it-1
21160 WRITE #1,array$(i,0),array$(i,1),array(i)
21190 NEXT i
21200 CLOSE 1
21210 RETURN

22000 ' *****
22010 ' Load
22020 ' *****
22030 PRINT cls$
22040 PRINT "Load data:" : PRINT
22050 INPUT "Name of file:",file$
22060 OPEN "i",1,file$
22070 INPUT #1,nn$,cu,it
22100 FOR i=0 TO cu-1
22110 INPUT #1,te$(i,0),te$(i,1),te(i)
22140 NEXT i
22150 FOR i=0 TO it-1
22160 INPUT #1,array$(i,0),array$(i,1),array(i)
22190 NEXT i
22200 CLOSE 1
22210 RETURN

23000 REM*****
23010 REM Error
23020 REM*****
23030 PRINT : PRINT CHR$(7) TAB(18) "***** " x$
      " *****"
23050 FOR i=1 TO 1000
23060 NEXT i
23070 RETURN
```

2.7.4. Binary search

For a full commentary on this module, see the equivalent module in Unifile. The only thing to note about the module is that the sorting is done on the basis of the item name in the zero column of ARRAY\$.

2.7.4.1. Lines 16000-16110

```

16000 ' *****
16010 ' Binary Search
16020 ' *****
16030 IF it=0 THEN ss=0:RETURN
16040 po=INT(LOG(it)/LOG(2)) : ss=2^po-1
16050 FOR i=po TO 0 STEP -1
16060   ss=ss+2^i*((array$(ss,0)>t1$)-(array$(ss,0)
      <t1$))
16070   IF ss<0 THEN ss=0
16080   IF ss>it-1 THEN ss=it-1
16090 NEXT i
16100 IF array$(ss,0)<t1$ THEN ss=ss+1
16110 RETURN

```

2.7.5. Inserting items into the main dictionary

The principle of this module is exactly the same as that of the parallel module in Unifile. The reason this module is slightly longer is that it has to insert two strings and a number into the two arrays rather than a single string.

2.7.5.1. Lines 17000-17110

```

17000 ' *****
17010 ' Insert Item
17020 ' *****
17030 FOR i=it TO ss+1 STEP -1
17040   array$(i,0)=array$(i-1,0)
17050   array$(i,1)=array$(i-1,1)
17060   array(i)=array(i-1)
17070 NEXT i
17080 array$(ss,0)=t1$
17090 array$(ss,1)=t2$
17100 array(ss)=nn
17110 RETURN

```

2.7.6. Entering items for the dictionary

Considerably less complicated than the equivalent module in Unifile, this module accepts three inputs from the user: a) the name of the item, b) the name of the units in which it is measured and c) the quantity associated with those units.

2.7.6.1. Lines 15000-15170

```
15000 ' *****
15010 ' Extend Dictionary
15020 ' *****
15030 WHILE 1
15040   PRINT cls$
15050   PRINT "New items for dictionary:":PRINT
15060   IF it>1000 THEN x$="NO MORE ROOM" : GOSUB
      23000 : RETURN
15070   q$="" : WHILE LOWER$(q$)<>"y"
15080     PRINT nn$; : INPUT " (name or 'l' to quit)
      : ",t1$
15090     IF t1$="l" THEN RETURN
15100     PRINT "Units"; : INPUT ":",t2$
15110     PRINT "Quantity per " LOWER$(t2$); :
      INPUT ":",nn
15120     PRINT : INPUT "Are these correct (y/n)";q$
      : PRINT
15130   WEND
15140   GOSUB 16000
15150   GOSUB 17000
15160   it=it+1
15170 WEND
```

2.7.7. The Search routine

As in Unifile, this module provides the user with the opportunity to move through the file of dictionary items, to search for named items or to delete items from the file. The module is simpler than that given in Unifile since it is designed to search for whole items only, rather than combinations of characters stored anywhere in an item. In addition, the structure of a complete entry in NNumber is far simpler than the structure of an item in the main array of Unifile.

2.7.7.1. Lines 18000-18250

```
18000 ' *****
18010 ' User Search
18020 ' *****
18030 ss=0
18040 t1$="":WHILE it>0
18050   PRINT cls$ : PRINT "Search:" : PRINT
18060   PRINT "Item number:" ss+1
```

```

18070 PRINT nn$ ":" array$(ss,0)
18080 PRINT "Units:" array$(ss,1)
18090 PRINT "Quantity per " LOWER$(array$(ss,1))
      ":" array(ss)
18100 PRINT : PRINT "Commands available:" :
      PRINT
18110 PRINT "Input item to be searched for"
18120 PRINT "ENTER for next item"
18130 PRINT "'#' then number to move pointer"
18140 PRINT "'d' to delete item"
18150 PRINT "'[' to quit"
18160 PRINT : INPUT "Which do you require";t1$
18170 IF t1$="" THEN t1$="#1"
18180 IF t1$="[" THEN RETURN
18190 IF LOWER$(t1$)="d" THEN GOSUB 190000 : t1$="
18200 IF LEFT$(t1$,1)="#" THEN ss=ss+VAL(MID$(t1$,
      ,2)) : t1$=""
18210 IF t1$<>"" THEN GOSUB 160000
18220 IF ss>it-1 THEN ss=it-1
18230 IF ss<0 THEN ss=0
18240 WEND
18250 RETURN

```

2.7.8. Deleting an item

The direct equivalent of the delete module in Unifile.

2.7.8.1. Lines 19000-19090

```

19000 ' *****
19010 ' Delete
19020 ' *****
19030 FOR i=ss TO it-2
19040   array$(i,0)=array$(i+1,0)
19050   array$(i,1)=array$(i+1,1)
19060   array(i)=array(i+1)
19070 NEXT i
19080 it=it-1
19090 RETURN

```

2.7.9. Copying items into the 'current' list

The purpose of NNumber is not simply to keep a dictionary of items and their associated quantities but to use that dictionary as the basis on which temporary lists, such as receipts, can be constructed. The modules which follow are therefore designed to allow the user to add items to the current list, to display that list, to delete single items from it or to delete the whole list in one operation. This module allows the copying of items from the main dictionary into the current list.

2.7.9.1. Lines 13000-13210

```
13000 ' *****
13010 ' Extend Current List
13020 ' *****
13030 WHILE 1
13040   PRINT cls$
13050   PRINT "Additions to current list:" : PRINT
13060   IF cu>100 THEN x$="CURRENT LIST NOW FULL" :
      GOSUB 23000 : RETURN
13070   PRINT nn$; : INPUT " ('[' to quit):",t1$
13080   IF t1$="[" THEN RETURN
13090   known=0 : GOSUB 16000 : IF array$(ss,0)=t1$
      THEN known=1
13100   IF known=0 THEN T$=UPPER$(nn$)+" UNKNOWN,
      PLEASE CHECK" : GOSUB 23000 : RETURN
13110   PRINT : PRINT "Units:" array$(ss,1)
13120   INPUT "Quantity:",q
13130   PRINT : INPUT "Are these correct (y/n)";q$
13140   WHILE LOWER$(q$)="y"
13150     te$(cu,0)=array$(ss,0)
13160     te$(cu,1)=MID$(STR$(q),2)+" "+array$(ss,1)
13170     te(cu)=q*array(ss)
13180     cu=cu+1
13190     q$=""
13200   WEND
13210 WEND
```

2.7.9.2. Commentary

Lines 13090-13100: These lines perform a check to see if the item input by the user, which is to be placed into the current list, is present in the main dictionary. This is done by calling up the binary search module to obtain the position at which the item would be inserted into the dictionary. The actual contents of

the dictionary at this point are then compared with what the user has input. If the item input is present in the dictionary, then the two items will be the same; otherwise an error message is printed and the module terminates.

Line 13110-13120: Having found the item in the dictionary, the module prints out the units in which it is normally measured and asks how many of those units are to be included.

Lines 13130-13200: The user is requested to confirm the accuracy of the entry before it is added to the current list, contained in the variables TE\$ and TE. Note that the quantity stored in TE is not the quantity per unit taken from the dictionary, but the *total* quantity for the number of units specified by the user.

2.7.10. Displaying the current list

The sole purpose of this module is to print the entries which make up the current list, one by one on the screen. After each entry, the user is required to press a key before the next is displayed. At the end of the list, the total of the associated quantities for the contents of the current list is given.

```
stock:Flanges
Units:5 Box
Quantity: 35
-----
stock:Widgets
Units:3 Bag
Quantity: 15
-----
stock:Widgets
Units:7 Bag
Quantity: 35
-----
Total: 85
Any key to return to menu
■
```

Drive is A:

Fig. 2-9: A current list on display

2.7.10.1. Lines 12000-12160

```
12000 ' *****
12010 ' Display Current List
12020 ' *****
12030 IF cu=0 THEN RETURN
12040 PRINT cls$ : ct=0
12050 FOR i=0 TO cu-1
12060   PRINT nn$ ":" te$(i,0)
12070   PRINT "Units:" te$(i,1)
12080   PRINT "Quantity:" te$(i)
12090   PRINT "-----"
12100   PRINT ""
12110   WHILE INKEY$="" : WEND
12120   ct=ct+te$(i)
12130 NEXT i
12140 PRINT : PRINT "Total:" ct
12150 PRINT : PRINT "Any key to return to menu"
12160 WHILE INKEY$="" : WEND
12170 RETURN
```

2.7.11. Search and delete from the current list

This is a much simplified version of the kind of search module used for the main dictionary.

2.7.11.1. Lines 20000-20170

```
20000 ' *****
20010 ' Current List Deletions
20020 ' *****
20030 i=0
20040 WHILE i<cu
20050   PRINT cls$
20060   PRINT "Current list deletions:" : PRINT
20070   PRINT te$(i,0)
20080   PRINT te$(i,1)
20090   PRINT : PRINT "Commands available:" :
20100   PRINT
20110   PRINT "ENTER for next item"
20120   PRINT "'d' to delete"
20130   PRINT "'\ ' to quit"
20140   PRINT : INPUT "Which do you require";q$
20150   IF q$="\ " THEN RETURN
```

```

20150 IF LOWER$(q$)<>"d" THEN i=i+1
20160 WHILE LOWER$(q$)="d"
20170   FOR j=i TO cu-1
20180     te$(j,0)=te$(j+1,0)
20190     te$(j,1)=te$(j+1,1)
20200     te(j)=te(j+1)
20210   NEXT j
20220   cu=cu-1
20230   q$="":WEND
20240 WEND
20250 RETURN

```

2.7.11.2. *Commentary*

Line 20160: If the user enters 'd', the current entry, which is indicated by the value of the variable I, is overwritten by copying subsequent items one place down the array.

2.7.12. Initialising the current list

For many applications, the need will be to construct a current list, obtain the total involved, and then quickly move on to a fresh list. This module wipes out the contents of the current list in one operation.

2.7.12.1. *Lines 14000-14090*

```

14000 ' *****
14010 ' Initialise Current List
14020 ' *****
14030 FOR i=0 TO cu-1
14040   te$(i,0)=""
14050   te$(i,1)=""
14060   te(i)=0
14070 NEXT i
14080 cu=0
14090 RETURN

```

2.8. Cardindx

We turn now to a long and relatively complex program which performs much the same function as the earlier Unifile, except that it brings to your filing needs all the space that can be provided on the disc drive. Whereas Unifile

worked with data in memory, only storing material on disc in between uses of the program, Cardindx keeps all its information on disc and is therefore not limited by the size of the memory. In making the jump to a disc based data system there is inevitably a slight loss of speed but that is more than compensated for by the increase in capacity. Which of the two programs suits your needs is something that only you can decide but they are both well worth having in your armoury.

Before proceeding to the program itself, a brief explanation of terminology cannot be avoided. When storing material in a disc file two terms are universally used to refer the structure of the material—'fields' and 'records'. A record is a collection of information placed onto a disc, such as the name, address and telephone number of an individual. A data file is a collection of records. Each record will normally be broken down into smaller units of information, the name, address, etc, referred to above, for instance. These smaller items are known as the 'fields' within the record. In commentary on the Unifile program we avoided these terms since there was little contact with disc files. In the course of Cardindx we shall be using the disc drive intensively and it would be a confusion not to use the same terms as you will find when you turn to your manual.

```

Card index amend data
Editing ADDBOOK

```

```

Name      Mr A. Jones
Company   Jones Builders Ltd.
Job       Director
Street    Jones Street
Town      Jones Town
County    Jones County
Code      JJJ BBB
Phone     999-888-777

```

```

Commands available:
1) First item      2) Next item
3) Last item       4) Previous item
5) Delete this item 6) Exit amend
Enter the command required ? █

```

Drive is A:

Fig. 2-10: Cardindx in data amendment mode

2.8.1. Initialisation

A standard initialisation routine with the addition of one or two lines to deal with the specific needs of a program which uses discs intensively.

2.8.1.1. Lines 10000-10120

```

10000 ' *****
10010 ' Setup
10020 ' *****
10030 CLEAR ,,,,416
10040 BUFFERS 8
10050 cls$ = CHR$(27)+"E"+CHR$(27)+"H"
10060 rvson$ = CHR$(27)+"p"
10070 rvsoff$ = CHR$(27)+"q"
10080 DEF FN locate$(x,y) = CHR$(27)+"Y"+CHR$(32+y)+CHR$(32+x)
10090 DIM rec$(9),buff$(8),prompt$(8)
10100 WIDTH 255,255
10110 ON ERROR GOTO 13000
10120 PRINT cls$

```

2.8.1.2. Commentary

Line 10030: This line clears the variables and sets the maximum size of a single record which can be written on the disc within the overall data file we shall be setting up.

Line 10040: The number of buffers, or areas of 128 bytes of memory to be set aside for the system to store parts of the index to the file we shall be using.

Line 10090: The main arrays used in the course of the program are:

REC\$: Used to store records input from the keyboard.

BUFF\$: Used to contain the individual fields into which a single record will be split when it is retrieved from the disc.

PROMPT\$: Used to hold the names of the individual fields into which an individual record can be split.

2.8.2. Print title

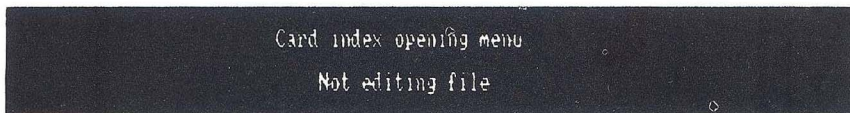
This short module turns a title, sent to it in the form of a string, into a properly centred inverse box on the screen. The title box will also inform the user whether the program is currently using a named data file.

2.8.2.1. Lines 27000-27120

```
27000 ' *****
27010 ' Print title
27020 ' *****
27030 PRINT FN locate$(1,1) rvson$
27040 FOR i = 1 TO 5
27050 PRINT FN locate$(1,i) SPC(79)
27060 NEXT i
27070 PRINT CHR$(27) "J"
27080 PRINT FN locate$(26,2) "Card index " title$
27090 PRINT FN locate$(30,4);
27100 IF file$="" THEN PRINT "Not editing file"
      ELSE PRINT "Editing " file$
27110 PRINT rvsoff$
27120 RETURN
```

2.8.3. Opening menu

A standard menu module which is used when the user first accesses the program. At this point, a decision has to be made as to whether the program is to be used to create a new data file, use an existing one or terminate the program. Unless one or other of the first two alternatives is chosen the main menu, which follows, will not be displayed.



Command available:

- 1) Create new card index
- 2) Open existing card index
- 3) Exit program

Enter command required ? █

Drive is A:

Fig. 2-11: The Cardindx opening menu

2.8.3.1. Lines 11000-11250

```

11000 ' *****
11010 ' Opening menu
11020 ' *****
11030 command = 0
11040 WHILE command<>3
11050   title$ = "opening menu"
11060   file$ = ""
11070   GOSUB 27000
11080   PRINT FN locate$(26,9) "Command available:"
11090   PRINT FN locate$(30,12) "1) Create new
      card index"
11100   PRINT FN locate$(30,14) "2) Open existing
      card index"
11110   PRINT FN locate$(30,16) "3) Exit program"
11120   t$=""
11130   WHILE t$<>"1" AND t$<>"2" AND t$<>"3"
11140     PRINT FN locate$(50,19) SPC(29) FN locate$
      (26,19);
11150     LINE INPUT "Enter command required ? ";t$
11160   WEND
11170   command = VAL(t$)
11180   ON command GOSUB 14000,15000
11190   IF file$<>" " THEN GOSUB 12000 : CLOSE 1
11200 WEND
11210 PRINT cls$ FN locate$(25,11);
11220 PRINT rvson$ SPC(21) rvsoff$
11230 PRINT TAB (26) rvson$ " Card index closed
      " rvsoff$
11240 PRINT TAB(26) rvson$ SPC(21) rvsoff$
11250 END

```

2.8.4. Main menu

Another standard menu module which becomes available once the program knows which file it is intended to be working with.

2.8.4.1. Lines 12000-12210

```

12000 ' *****
12010 ' Main menu
12020 ' *****
12030 command = 0

```

```
12040 WHILE command<>4
12050   title$ = "main menu"
12060   GOSUB 27000
12070   PRINT FN locate$(26,9) "Commands available:"
12080   PRINT FN locate$(30,12) "1) Add new cards"
12090   PRINT FN locate$(30,14) "2) Examine/Modify
      current cards"
12100   PRINT FN locate$(30,16) "3) Print out this
      card index"
12110   PRINT FN locate$(30,18) "4) Return to main
      menu"
12120   t$ = ""
12130   WHILE t$<>"1" AND t$<>"2" AND t$<>"3" AND
      t$<>"4"
12140     PRINT FN locate$(50,21) SPC(29) FN locate$
      (26,21);
12150     LINE INPUT "Enter the command required ? "
      ;t$
12160   WEND
12170   command = VAL(t$)
12180   ON command GOSUB 17000,20000,26000
12190 WEND
12200 command = 0
12210 RETURN
```

2.8.5. Error trapping

One of the more nerve wracking features about working with data directly on disc, is that if the program should go wrong (and all programs do go wrong at some time) you are in danger of corrupting important material. The solutions are twofold. First, if data is of any importance at all, you should always keep an up-to-date copy of the file in which it is stored and secondly, you should try to ensure that if the program does go wrong it stops itself in an orderly way, without leaving an incomplete file on the disc whose contents you will have difficulty accessing in future. The sole purpose of this module is to mark the current data file as being valid and close it properly before terminating the program and displaying the error message which caused the problem.

2.8.5.1. Lines 13000-13070

```
13000 ' *****
13010 ' Error trapping
13020 ' *****
13030 PRINT cls$ rvsoff$ CHR$(7)
```

```

13040 t = CONSOLIDATE(1)
13050 CLOSE 1
13060 ON ERROR GOTO 0
13070 END

```

2.8.6. Ask user to confirm input

At several points during the course of the program, Cardindx will ask the user to confirm the contents of important inputs. Rather than scatter lines around to fulfill this task it is far easier to leave it to a single module which is designed to do the job thoroughly.

2.8.6.1. Lines 29000-29140

```

29000 ' *****
29010 ' Ask user to confirm input
29020 ' *****
29030 t$ = ""
29040 PRINT CHR$(27) "j";
29050 WHILE t$<>"Y" AND t$<>"N"
29060   PRINT CHR$(27) "k (Y/N) ? " CHR$(27) "D";
29070   WHILE t$<>"Y" AND t$<>"N"
29080     t$ = UPPER$(INPUT$(1))
29090     IF t$="Y" OR t$="N" THEN PRINT t$;
29100   WEND
29110   IF INPUT$(1)<>CHR$(13) THEN t$ = ""
29120 WEND
29130 ok = t$="Y"
29140 RETURN

```

2.8.6.2. Commentary

Line 29130: The result of the module is returned to the rest of the program in the form of the variable OK. If whatever piece of information the module was being applied to is confirmed by the user, then the value of OK will be set to -1—otherwise the calling module knows that the user has registered an error and will take the appropriate action.

2.8.7. Get file name from user

Using the Cardindx program requires more access to disc files than previous programs, so it is sensible to have a separate module which allows the user to input a name—then checking to make sure it is valid before returning the name to the calling module.

2.8.7.1. Lines 28000-28190

```
28000 ' *****
28010 ' Get file name from user
28020 ' *****
28030 ok = 0
28040 WHILE NOT ok
28050   file$ = ""
28060   GOSUB 27000
28070   PRINT FN locate$(1,12) ;
28080   LINE INPUT "Enter new file name ? ";file$
28090   file$ = UPPER$(file$)
28100   ok = LEN(file$)<=8
28110   FOR i = 1 TO LEN(file$)
28120     ok = ok AND MID$(file$,i,1)>="A" AND MID$
      (file$,i,1)<="Z"
28130   NEXT i
28140   PRINT FN locate$(1,12) "You entered
      " file$
28150   IF NOT ok THEN PRINT : PRINT CHR$(7) "
      Illegal name - press a key" : t$=INPUT$(1) :
      GOTO 28180
28160   PRINT FN locate$(1,16) "Is this correct ";
28170   GOSUB 29000
28180 WEND
28190 RETURN
```

2.8.7.2. Commentary

Lines 28110-28130: The major check that the module makes, apart from asking the user to confirm the name, is to check to see that it consists only of allowable characters, ie, that the system will not throw out the file name as invalid.

2.8.8. Create new file

This module accepts from the user the information to set up a new data file. Cardindx differs from Unifile in that not only must the number of fields in every record in the file be specified, so must their maximum length. The maximum length of a single field with the program is 50 characters, and every line will be the same length. This does not mean that the user has to type in 50 characters, only that Cardindx will store the field in a space on the disc which is sufficient for that number. The maximum number of lines on a single card, or record, is eight.

2.8.8.1. *Lines 14000-14330*

```

14000 ' *****
14010 ' Create new file
14020 ' *****
14030 title$ = "new card index"
14040 GOSUB 28000
14050 ok = 0
14060 WHILE NOT ok
14070   GOSUB 27000
14080   PRINT FN locate$(0,7) ;
14090   LINE INPUT "Number of characters per line (
1-50) ? ";c$
14100   c = VAL(c$)
14110   IF c<1 OR c>50 THEN PRINT : PRINT CHR$(7) "
Illegal value" : GOTO 14240
14120   PRINT
14130   LINE INPUT "Number of lines per card (1-8)
? ";l$
14140   l = VAL(l$)
14150   IF l<1 OR l>8 THEN PRINT : PRINT CHR$(7) "
Illegal value" : GOTO 14240
14160   PRINT
14170   FOR i = 1 TO l
14180     PRINT "Enter prompt for line" i "(7
characters max.) ? ";
14190     LINE INPUT prompt$(i)
14200     prompt$(i) = LEFT$(prompt$(i)+SPACE$(7),7)
14210   NEXT i
14220   PRINT
14230   PRINT "Is this correct"; : GOSUB 29000
14240 WEND
14250 GOSUB 27000
14260 PRINT FN locate$(20,12) "Creating new card
index - please wait . . ."
14270 t$ = DEC$(1,"##")+DEC$(c,"##")
14280 FOR i = 1 TO l
14290   t$ = t$+prompt$(i)
14300 NEXT i
14310 CREATE 1,file$+".dta",file$+".idx",0,l*(c+2)
,t$
14320 ON 1 GOSUB 16030,16040,16050,16060,16070,
16080,16090,16100
14330 RETURN

```

2.8.8.2. *Commentary*

Lines 14090-14110: The user is requested to input the length of the lines on the individual card. If a value outside the range 1-50 is input, an error message is displayed and the main loop is repeated.

Lines 14130-14150: The user is requested to input the number of lines on the card. If a value outside the range 1-8 is input, an error message is displayed and the main loop is repeated.

Lines 14170-14210: If the figures for the number characters and number of lines are valid, the user is requested to supply a prompt to be attached to each line when an record is being made to the file. Note that the prompts are placed starting at position one in the array, not zero. Later on you will find that the spare line of the array at zero makes it possible to use temporary prompts by storing the contents of line one in line zero and placing the temporary prompt into line one.

Lines 14270-14300: The number of characters per line, the number of lines and the prompts, each padded out to seven characters if necessary, are all stored in the variable T\$.

Line 14310: The data file is created, together with a parallel index file. The temporary variable T\$ is written into the data file in the form of a 'user string', a string which can be written into or read from the file before the program gets down to the job of formally accessing the file. The record length which the system will use when picking up a single record from the file, is set to the number of lines multiplied by the number of characters per line plus two.

Line 14320: The next module, which sets up the structure of the individual fields, or fields within a single record, is called. The part of the module to be executed depends on the number of lines on a single card.

2.8.9. All field statements for $l = 1$ to 8

When setting up a data file composed of records (a whole record) and fields (the fields within an record), it is necessary to spell out in full the length of each field and to give the system the name of a variable into which the contents of the field can be placed when a particular record is picked up from the data file on disc. If the number of fields you wish to set up varies, the only way in which the process can be carried out in a single program is to have a number of lines, each specifying a different number of fields be set up. This module

therefore contains eight different lines coping with the possible choices of one to eight fields on a single card.

2.8.9.1. Lines 16000-16100

```

16000 ' *****
16010 ' All field statements for l=1 to 8
16020 ' *****
16030 FIELD #1,c AS buff$(1) : RETURN
16040 FIELD #1,c AS buff$(1),c AS buff$(2):
      RETURN
16050 FIELD #1,c AS buff$(1),c AS buff$(2),c AS
      buff$(3) : RETURN
16060 FIELD #1,c AS buff$(1),c AS buff$(2),c AS
      buff$(3),c AS buff$(4) : RETURN
16070 FIELD #1,c AS buff$(1),c AS buff$(2),c AS
      buff$(3),c AS buff$(4),c AS buff$(5) :
      RETURN
16080 FIELD #1,c AS buff$(1),c AS buff$(2),c AS
      buff$(3),c AS buff$(4),c AS buff$(5),c AS
      buff$(6) : RETURN
16090 FIELD #1,c AS buff$(1),c AS buff$(2),c AS
      buff$(3),c AS buff$(4),c AS buff$(5),c AS
      buff$(6),c AS buff$(7) : RETURN
16100 FIELD #1,c AS buff$(1),c AS buff$(2),c AS
      buff$(3),c AS buff$(4),c AS buff$(5),c AS
      buff$(6),c AS buff$(7),c AS buff$(8) :
      RETURN

```

2.8.9.2. Commentary

Lines 16030-16100: Each of these lines tells the system that file number one, which has already been created by the previous module, is to have one or more fields of C characters and that the contents of that field are to be placed into the corresponding element of the array BUFF\$ whenever a record is pulled back from the data file on disc.

2.8.10. Open existing file

Opening a data file which already exists on the disc is a much simpler matter than creating one from scratch, as this module illustrates.

2.8.10.1. Lines 15000-15120

```
15000 ' *****
15010 ' Open existing file
15020 ' *****
15030 title$ = "open existing card index"
15040 GOSUB 28000
15050 OPEN "k",1,file$+".dta",file$+".idx",0,,t$
15060 l = VAL(MID$(t$,1,2))
15070 c = VAL(MID$(t$,3,2))
15080 FOR i = 1 TO l
15090   prompt$(i) = MID$(t$,i*7-2,7)
15100 NEXT i
15110 ON l GOSUB 16030,16040,16050,16060,16070,
      16080,16090,16100
15120 RETURN
```

2.8.10.2. Commentary

Line 15050: The file is opened for access, the 'k' indicating to the system that the file is keyed, ie, it has an index file associated with it. The variable T\$, which contains the length and number of lines plus the prompts to be attached to each line, is recovered from the file in the OPEN statement itself.

Lines 15060-15070: The line length and number of lines are re-instated in the appropriate variables, C and L.

Lines 15080-15100: The seven character prompts to be attached to each line of a single record are extracted from T\$ and placed into PROMPT\$.

Line 15110: As with the creation of a new file, the field structure must be set with a single line, which varies according to the number of fields to be declared.

2.8.11. Add new record to file

This task of this short module is little more than to control the flow of the program between the two subroutines for input of records and writing an record to disc, which follow.

2.8.11.1. Lines 17000-17100

```

17000 ' *****
17010 ' Add new record to file
17020 ' *****
17030 title$ = "Add new cards"
17040 FOR i = 1 TO 1
17050   rec$(i) = SPACE$(c)
17060 NEXT i
17070 rec$(1+1) = ""
17080 GOSUB 18000
17090 GOSUB 19000
17100 RETURN

```

2.8.11.2. Commentary

Lines 17040-17070: The array REC\$, used to store the record as it is input, is padded out to the correct number of characters per line.

2.8.12. Input record from keyboard

We now turn to the problem of inputting a record from the keyboard. The length of this module is a reflection of the fact that data is input in a highly formatted way, with the screen laid out in the form of a card. A great deal of the work of this module is spent on controlling the cursor and allowing the user to move around the surface area of the card being input.

2.8.12.1. Lines 18000-18450

```

18000 ' *****
18010 ' Input record from keyboard
18020 ' *****
18030 ok = 0
18040 WHILE NOT ok
18050   GOSUB 27000
18060   PRINT rvson$
18070   i = 1
18080   WHILE rec$(i) <> ""
18090     PRINT FN locate$(1,i+10) prompt$(i)
18100     PRINT FN locate$((80-c)/2,i+10) rec$(i)
18110     i = i+1
18120   WEND
18130   vp = 1

```

```
18140 hp = 1
18150 WHILE rec$(vp)<>" "
18160 PRINT FN locate$((80-c)/2-1+hp, vp+10) ;
18170 t$ = INPUT$(1)
18180 t = ASC(t$)
18190 IF t=1 AND hp>1 THEN hp = hp-1
18200 IF t=6 AND hp<=LEN(rec$(vp)) THEN hp = hp+1
18210 IF t=13 THEN vp = vp+1 : hp = 1
18220 IF t=30 THEN vp = vp+1
18230 IF t=31 AND vp>1 THEN vp = vp-1
18240 WHILE t=127 AND hp>1
18250 t = 0
18260 t1$ = ""
18270 t2$ = ""
18280 hp = hp-1
18290 IF hp>1 THEN t1$ = LEFT$(rec$(vp), hp-1)
18300 IF hp<c THEN t2$ = MID$(rec$(vp), hp+1)
18310 rec$(vp) = t1$+t2$+" "
18320 PRINT CHR$(8) t2$ " "
18330 WEND
18340 WHILE t>=32 AND t<127 AND hp<=LEN(rec$(vp))
18350 MID$(rec$(vp), hp, 1) = t$
18360 PRINT t$;
18370 t=0
18380 hp = hp+1
18390 WEND
18400 WEND
18410 PRINT rvsoff$
18420 PRINT FN locate$(1,20) "Is this correct" ;
18430 GOSUB 29000
18440 WEND
18450 RETURN
```

2.8.12.2. *Commentary*

Lines 18080-18120: The existing card is displayed on the screen, including its prompts. On the first display the card will be blank but if the main loop is repeated, these lines will display the current contents.

Lines 18130-18140: These two variables record the vertical (VP) and horizontal (HP) position of the cursor within the display of the card.

Line 18150: This loop will continue to execute until the cursor moves onto a line of REC\$ which is empty. What this really means is that the loop will continue until the user press RETURN sufficient times to move the cursor off the last line of the card—in the previous module the next line of REC\$ was set to a null string, ie, nothing.

Lines 18160: The cursor is positioned at the point indicated by VP and HP.

Lines 18170-18180: The input of a single key is accepted from the keyboard and translated into its ASCII code.

Lines 18190-18230: These lines control the movement of the cursor by means of the cursor keys and the RETURN key. According to the key pressed, the value of either HP, VP or both, is altered, with protection against moving the cursor outside the limits of the card, except at the bottom.

Lines 18240-18330: These lines deal with the use of the <-DEL key. When this key is detected, the current contents of the line of REC\$ on which the cursor sits are split into two separate strings, one which ends just before the character to be deleted and another which starts just after it. The two strings are then combined to form a line excluding the character to be deleted.

Lines 18340-18390: These lines cope with normal characters, ie, those which are not cursor characters, RETURN or DELeTe. When such a key is detected, the character it represents is added to the relevant line of REC\$, the line is reprinted and the cursor moved one place to the right.

2.8.13. Write record to file

Having given ourselves the ability to enter the data for a card, we now turn to placing the data into the disc file. The facilities provided for file handling in Mallard Basic make the task relatively simple compared to other forms of the language. In adding the record to the file, we do not need to bother with clever insertion techniques to find the correct position in the file. The ordering of the file will depend not on the order of records in the main file itself but on the record of the position of records contained in the key, or index, file which will be added to at the same time as the main data is stored.

The structure of the index file will be quite different from that of the main file. What it will contain is the whole (or extracts if they are too big) of the fields from each record which we have specified to the system we want to index.

Fields of the same rank, ie, all the items which occupy field one in a record, will be indexed together. The order in which the rank will be indexed will depend on the command issued to the system when the file was set up, since it is possible to have any field as the primary one for indexing purposes.

The illustration gives a rough idea of the relationship between the main data file and the index file:

Data file

Index file

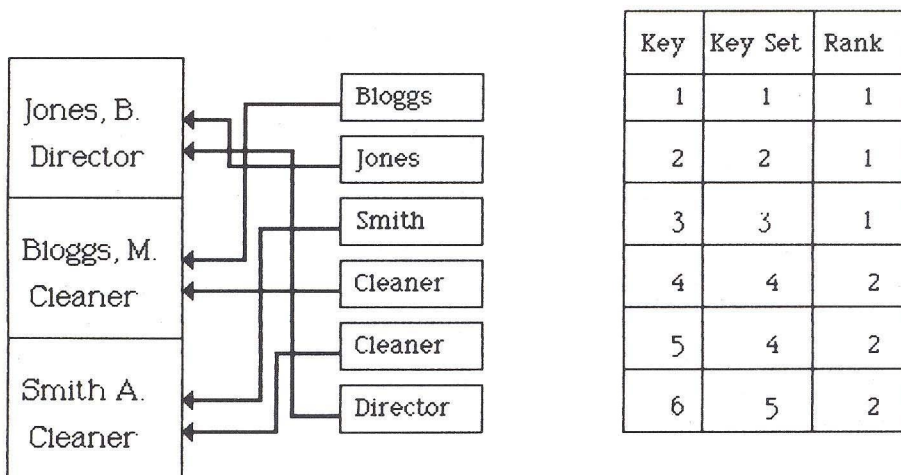


Fig. 2-12: Indexing a simple file

The boxes on the left represent the main data file, with each record having two fields, one for the name and one for the job. The file has been set up to index first on field one and secondly on field two. In the index file, therefore, we find the names, in alphabetical order, followed by the job specifications, also in order. Attached to each index entry are three values recording:

- The position of the key in the index file.
- The 'key set'. This is another way of expressing the position of the key in the index file, but this time counting any groups of identical keys as a single entry—thus the two cleaners are part of the same key set.
- The rank, representing the field number from which the contents of the key was taken.

The arrows connecting the keys to the original item are a reminder that each key also stores the position in the main file of the record from which it was taken.

2.8.13.1. *Lines 19000-19190*

```

19000 ' *****
19010 ' Write record to file
19020 ' *****
19030 FOR i=1 TO 1
19040 LSET buff$(i) = rec$(i)
19050 NEXT i
19060 e = ADDREC(1,0,0,UPPER$(LEFT$(rec$(1),31)))
19070 i = 1
19080 WHILE e=0 AND i<1
19090 e = ADDKEY(1,0,i,UPPER$(LEFT$(rec$(i+1),31)
    ),FETCHREC(1))
19100 i = i+1
19110 WEND
19120 WHILE e<>0
19130 PRINT FN locate$(1,22) CHR$(7) "*****
    ERROR in writing record to file *****"
19140 PRINT " Press a key . . ."
19150 t$ = INPUT$(1)
19160 e = 0
19170 WEND
19180 e = CONSOLIDATE(1)
19190 RETURN

```

2.8.13.2. *Commentary*

Lines 19030-19050: The contents of REC\$, representing the data for the new card, are written into BUFF\$, the special variable where the system has been told to expect to find the contents of the fields making up a record. Note the use of LSET here—it is not possible to simply use '=' to set up the contents of the fields.

Line 19060: The new record is added to the file. This line automatically inserts the contents of BUFF\$ into the file, without having to supply the name of the array—it has already been nominated as the array containing the fields making up the new record. The reference to REC\$(1) at the end of the line places the first 31 characters of the first line of REC\$ into the key file. The system automatically stores in the key file the location at which the record with that key is to be found in the main file.

A secondary effect of the line is to obtain from the system an error level number which will be stored in the variable E. This should normally be zero, indicating that no problem has occurred. If something has gone wrong, however, such as a disc change (meaning that the system cannot find the right file), E will be set to a level which will notify the program of the problem.

Lines 19070-19110: So far, what we have done is to store a new record and store in the index file a reference to the first field in that new record. What these lines do is to add to the index file a series of separate keys, one for each field in the record. Later on in the use of the program, we shall be able to search for records of the basis of the contents of fields other than the first. Once again, the variable E is used to store any error messages returned by the system during the process.

Lines 19120-19170: If the error variable E indicates that a problem has been encountered by the system, an error message is generated.

Line 19180: Having changed the file, the system regards the main data file and the index file as possibly being inconsistent with each other. Applying the CONSOLIDATE command ensures that any outstanding material in buffers in memory is written to the files and they are marked as ready for use.

2.8.14. Search file for record

We now turn to a series of interlocking modules which allow the user to retrieve material from the file in a variety of ways, based on the information contained in the key file. This first module is, in many ways, little more than a complex menu which allows the user to specify what the program is to search for, in choosing the records to be displayed.

2.8.14.1. Lines 20000-20880

```
20000 ' *****
20010 ' Search file for record
20020 ' *****
20030 title$ = "user search"
20040 GOSUB 27000
20050 PRINT FN locate$(26,12) "Commands available:
20060 PRINT FN locate$(30,14) "1) Search for word
      or phrase"
20070 PRINT FN locate$(30,16) "2) Search for
      complete item"
```

```

20080 PRINT FN locate$(30,18) "3) Search every
      card"
20090 PRINT FN locate$(26,20) "Enter the command
      required ? ";
20100 t$ = ""
20110 WHILE t$<>"1" AND t$<>"2" AND t$<>"3"
20120   PRINT CHR$(8) " " CHR$(8);
20130   WHILE t$<>"1" AND t$<>"2" AND t$<>"3"
20140     t$ = INPUT$(1)
20150   WEND
20160   PRINT t$;
20170   IF INPUT$(1)<>CHR$(13) THEN t$ = ""
20180 WEND
20190 key$ = ""
20200 f = 1
20210 WHILE t$="1" OR t$="2"
20220   t1$ = t$
20230   rec$(1) = SPACE$(c)
20240   rec$(2) = ""
20250   prompt$(0) = prompt$(1)
20260   IF t$="1" THEN prompt$(1) = "Phrase" ELSE
      prompt$(1) = "Item"
20270   GOSUB 18000
20280   prompt$(1) = prompt$(0)
20290   key$ = rec$(1)
20300   WHILE LEFT$(key$,1)=" " AND t1$="1"
20310     key$ = MID$(key$,2)
20320   WEND
20330   WHILE RIGHT$(key$,1)=" " AND t1$="1"
20340     key$ = MID$(key$,1,LEN(key$)-1)
20350   WEND
20360   t$ = ""
20370 WEND
20380 GOSUB 27000
20390 PRINT FN locate$(26,10) "Select field to
      search:"
20400 PRINT FN locate$(30,12) "0) All fields"
20410 FOR i = 1 TO 1
20420   PRINT FN locate$(30,12+i) CHR$(i+48) " ) "
      prompt$(i)
20430 NEXT i
20440 PRINT FN locate$(26,13+i) "Enter option
      required ? ";

```



```
20450 t$ = ""
20460 WHILE t$<"0" OR t$>CHR$(1+48) OR LEN(t$)<>1
20470 PRINT CHR$(8) " " CHR$(8);
20480 WHILE t$<"0" OR t$>CHR$(i+48) OR LEN(t$)<>1
20490 t$ = INPUT$(1)
20500 WEND
20510 PRINT t$;
20520 IF INPUT$(1)<>CHR$(13) THEN t$ = ""
20530 WEND
20540 f = VAL(t$)
20550 IF key$="" THEN ig = 0 : GOTO 20600
20560 GOSUB 27000
20570 PRINT FN locate$(1,12) "Should the search
      ignore case" ;
20580 GOSUB 29000
20590 ig = ok
20600 title$ = "amend data"
20610 GOSUB 27000
20620 GOSUB 21000
20630 IF NOT found THEN e = 9999
20640 WHILE e=0
20650 FOR i = 1 TO 1
20660 PRINT FN locate$(1,7+i) rvson$ prompt$(i)
      rvsoff$ " " rvson$ buff$(i) rvsoff$
20670 NEXT i
20680 PRINT
20690 PRINT TAB(9) "Commands available:"
20700 PRINT TAB(12) "1) First item          2)
      Next item"
20710 PRINT TAB(12) "3) Last item          4)
      Previous item"
20720 PRINT TAB(12) "5) Delete this item    6)
      Exit amend"
20730 PRINT TAB(9) "Enter the command required ?
      ";
20740 t$ = ""
20750 WHILE t$<"1" OR t$>"6" OR LEN(t$)<>1
20760 PRINT CHR$(8) " " CHR$(8);
20770 WHILE t$<"1" OR t$>"6" OR LEN(t$)<>1
20780 t$ = INPUT$(1)
20790 WEND
20800 PRINT t$;
20810 IF INPUT$(1)<>CHR$(13) THEN t$ = ""
```

```

20820 WEND
20830 command = VAL(t$)
20840 ON command GOSUB 21000,22000,24000,23000,
      25000
20850 IF command>=5 THEN e = 9999 ELSE e = 0
20860 WEND
20870 command = 0
20880 RETURN

```

2.8.14.2. *Commentary*

Lines 20030-20180: The opening menu for the module. The options available are to search for a word or phrase (ie, any combination of characters), a complete field within a record or to page through the file in index order. Only the characters 1 to 3 are accepted.

Lines 20210-20370: If the user specifies that a word or phrase is to be searched for, the input module at 18000 is used to allow input of the target characters by setting up, in effect, a one line card with the prompt 'Phrase' or 'Item'. Any leading or trailing spaces are stripped off. Note the way in which the prompt to be used is placed into line one of PROMPT\$, with the current contents being stored temporarily in line zero, which is not used by the input routine. The word or phrase to be searched for is stored in the variable KEY\$.

Lines 20380-20540: A second menu which requires the user to specify which of the fields within each record is to be examined during the search. The characters accepted are zero to the number of the last field in the record, with zero meaning that every field is to be examined. The number of the field to be searched is stored in the variable F.

Line 20550: The variable IG is used to tell the program whether it is to ignore the case of the characters when conducting the search, so that 'abc' will be counted as equivalent to 'ABC'. This particular line applies only to the 'Search every card' option, where there is no key to be searched for.

Lines 20560-20590: If there is a phrase or word to be searched for, the user is required to specify whether the case of the letters is to be ignored in conducting the search. The result is again stored in the variable IG.

Lines 20600-20630: The search for the word or phrase specified is instituted by a call to the module at line 21000. If the target is not found, this will be indicated by the value of the variable FOUND and the error variable E will be set to a unique value of 9999 and the user will be returned to the main menu.

Lines 20640-20860: The main menu for the module, which only appears when the first field satisfying the search criteria has been found. The options in the menu are all based on the same search target, which can only be changed by quitting the module for the main program menu and starting it up again. Provided that the user does wish to continue with the same target, the options are to go to the first or last field containing the target, the previous or next field containing it, to delete the current record or to exit from the module back to the main menu. The search options are all based on modules which you have yet to enter.

2.8.15. Find first record matching key

A short module which does little more than call the next one. The module applies only to a search for the first record in the file which contains the search target.

2.8.15.1. Lines 21000-21070

```
21000 ' *****
21010 ' First item matching key
21020 ' *****
21030 IF f=0 THEN e = SEEKRANK(1,0,0) ELSE e =
      SEEKRANK(1,0,f-1)
21040 first = -1
21050 GOSUB 22000
21060 first = 0
21070 RETURN
```

2.8.15.2. Commentary

Line 21030: The system is set to point to the first entry in the index file of the correct rank. It is worth remembering that all the commands which begin SEEK are different ways of moving around the index file to find index entries which conform to some criterion or other. In each case, though the program is not aware of it, the system will remember, until moved in some way, the number of the index entry and of the record in the main file which is associated with it. In the next module you will meet the two FETCH commands used to get these values from the system and store them safely in variables.

2.8.16. Next record matching key

This module and the next are the two workhorses when searching through a data file for a search target. The purpose of this module is to search through

the file for the next record containing the search target in the right field. The module passes the value E to whatever part of the program that calls it, representing possible error messages from the system and FOUND, which will be either -1 or zero, depending on whether the field being searched for has been found or not.

2.8.16.1. Lines 22000-22210

```

22000 ' *****
22010 ' next item matching key
22020 ' *****
22030 IF f=0 THEN rank=0 ELSE rank = f-1
22040 lastkey$ = FETCHKEY$(1)
22050 lastrec = FETCHREC(1)
22060 e = SEEKNEXT(1,0)
22070 IF first OR e=102 OR e=103 THEN e = SEEKREC(
1,0,rank,lastkey$,lastrec)
22080 found = 0
22090 WHILE (e=0 OR e=101) AND NOT found
22100   GET #1
22110   IF f=0 THEN field1 = 1 : field2= 1 ELSE
field1 = f : field2 = f
22120   IF ig THEN key$ = UPPER$(key$)
22130   FOR i = field1 TO field2
22140     IF ig THEN t$ = UPPER$(buff$(i)) ELSE t$ =
buff$(i)
22150     found = found OR (INSTR(t$,key$)>0)
22160   NEXT i
22170   IF NOT found THEN e = SEEKNEXT(1,0)
22180 WEND
22190 IF NOT found THEN e = SEEKREC(1,0,rank,
lastkey$,lastrec) : GET #1
22200 IF e=101 THEN e=0
22210 RETURN

```

2.8.16.2. Commentary

Line 22030: The variable F will contain zero if every field is to be searched or the value of a particular field if only one field is to be searched.

Lines 22040-22050: The number of the last record and key accessed is stored in the variables LASTREC and LASTKEY.

Lines 22060-22070: SEEKNEXT is used to find the next entry in the index file. A variety of information about the next entry is supplied by the system and stored in the variable E. The values we are interested in are:

102: indicates that the next entry in the index file has been found but that it has a different rank to the last one (ie, it refers to a different field).

103: there are no further entries in the index file.

Whichever of these messages is found, we do not wish to move on to the next index entry. In one case it doesn't exist and in the other there is no field of the correct rank that the user has specified for the search. In either case, the system is called back to point to the last record accessed.

The line is also executed if the variable FIRST indicates that what is being searched for is the first record which conforms to the search pattern laid down. In this case the first record with a field of the correct rank has already been found and this module is only being called to examine the record.

Lines 22080-22180: Provided that the message returned from the system is either zero or 101, the search is carried out. These error messages both indicate that the next index entry has been found and that it is of the same rank. The record is taken into memory and automatically stored in BUFF\$\$ by the GET # command. The variables FIELD1 and FIELD2 are set to indicate whether all the fields in the record are to be searched or only one, then depending on the value of the variable IG, the target string is compared with the contents of the fields between FIELD1 and FIELD2, either exactly as the user has entered them or with both set to upper case so that any differences in the case of the characters are eliminated.

If a match between the target and the contents of the field is discovered, the variable FOUND is set to -1, otherwise the SEEKNEXT command is used again to move on to the next index entry. The loop will terminate when FOUND indicates that a match has been discovered or when SEEKNEXT informs the program that there is not another key of the appropriate rank in the file.

Line 22190: At this point the search has been terminated. If a field has not been found, as indicated by the value of FOUND, then the system is reset to point to the record whose number was stored in LASTKEY and LASTREC at the beginning of the module.

Line 22200: The error message 101 is irrelevant to us for the present purposes, indicating that another index entry has been found of the same rank but that its contents are different from the last entry. For our purposes this is perfectly satisfactory and we may as well simplify decisions elsewhere in the program by simply setting 101 to zero.

2.8.17. Previous item matching key

This module is the mirror image of the previous one. The only difference is that instead of using the SEEKNEXT command, the SEEKPREV command is used to find the *previous* index entry of the correct rank.

2.8.17.1. Lines 23000-23210

```

23000 ' *****
23010 ' Previous item matching key
23020 ' *****
23030 IF f=0 THEN rank=0 ELSE rank = f-1
23040 lastkey$ = FETCHKEY$(1)
23050 lastrec = FETCHREC(1)
23060 e = SEEKPREV(1,0)
23070 IF first OR e=102 OR e=103 THEN e = SEEKREC(
    1,0,rank,lastkey$,lastrec)
23080 found = 0
23090 WHILE (e=0 OR e=101) AND NOT found
23100   GET #1
23110   IF f=0 THEN field1 = 1 : field2 = 1 ELSE
    field1 = f : field2 = f
23120   IF ig THEN key$ = UPPER$(key$)
23130   FOR i = field1 TO field2
23140     IF ig THEN t$ = UPPER$(buff$(i)) ELSE t$ =
    buff$(i)
23150     found = found OR (INSTR(t$,key$)>0) OR
    key$=""
23160   NEXT i
23170   IF NOT found THEN e = SEEKPREV(1,0)
23180 WEND
23190 IF NOT found THEN e = SEEKREC(1,0,rank,
    lastkey$,lastrec) : GET #1
23200 IF e=101 THEN e = 0
23210 RETURN

```

2.8.18. Last item matching key

Just as a simple module was all that was needed to use the search facilities to find the first record corresponding to the search target, so this short one finds the last record. This one is slightly longer, since before calling up the previous module, a loop must be used to move through the file to the last record with an index entry of the correct rank. There is no command corresponding to SEEKRANK which will automatically find the last appropriate entry.

2.8.18.1. Lines 24000-24150

```
24000 ' *****
24010 ' Last item matching key
24020 ' *****
24030 IF f=0 THEN rank = 0 ELSE rank = f-1
24040 e = SEEKRANK(1,0,rank)
24050 WHILE e=0
24060   lastkey$ = FETCHKEY$(1)
24070   lastrec  = FETCHREC(1)
24080   e = SEEKNEXT(1,0)
24090   IF e=101 THEN e=0
24100 WEND
24110 IF e=102 OR e=103 THEN e = SEEKREC(1,0,rank,
      lastkey$,lastrec)
24120 first = -1
24130 GOSUB 23000
24140 first = 0
24150 RETURN
```

2.8.18.2. Commentary

Line 24040: The first index entry of the correct rank is located.

Lines 24050-24100: This loop constantly stores the location of the index and main data file record with a field of the appropriate rank and then looks for the next. When the variable E indicates that the system cannot find another valid entry in the index file, the variables LASTKEY\$ and LASTREC will contain the details of the last record in the file.

Line 24110: The system is set to point to the record indicated by LASTKEY\$ and LASTREC.

Lines 24120-24140: The previous module is called to carry out the task of working backwards through the file to find the last record containing the search target.

2.8.19. Delete record

The final module to be added to the search facilities is deletion. This is slightly more complex than might be imagined because the way in which fields are deleted from a data file is to delete the index entries associated with them. The index entries for a particular record in the data file are, of course, distributed through the index file according to rank, and within rank in alphabetical order. In other words, the index entries are not in one identifiable place. The main part of the module is therefore devoted to searching out index entries which appear to refer to the record to be deleted, checking that they do in fact refer to that record and then deleting them. When the last key referring to the record is deleted, so is the record itself.

2.8.19.1. Lines 25000-25200

```

25000 ' *****
25010 ' Delete record
25020 ' *****
25030 FOR i = 1 TO 1
25040   rec$(i) = buff$(i)
25050 NEXT i
25060 FOR rank = 0 TO 1-1
25070   e = SEEKKEY(1,0,rank,LEFT$(UPPER$(rec$(
      rank+1)),31))
25080   ok = 0
25090   WHILE e=0 AND NOT ok
25100     GET #1
25110     ok = -1
25120     FOR i = 1 TO 1
25130       ok = ok AND (rec$(i)=buff$(i))
25140     NEXT i
25150     IF NOT ok THEN e = SEEKNEXT(1,0)
25160   WEND
25170   IF ok THEN e = DELKEY(1,0)
25180 NEXT rank
25190 e = CONSOLIDATE(1)
25200 RETURN

```

2.8.19.2. Commentary

Lines 25030-25050: The contents of BUFF\$, the record to be deleted, are copied into REC\$.

Lines 25060-25180: For each line of the record to be deleted, this loop searches out the first index entry to match it. It is possible that the index entry merely points to a similar field in the wrong record, so the second loop compares every field stored in REC\$ with every field in the current record in BUFF\$. If the two are identical then the index entry is deleted. If the two are not identical, then SEEKNEXT is used to find another index entry and the process of comparison is repeated.

At each repetition of the main loop, the index entry for one of the fields in the record to be deleted is removed. When the last index entry is removed, so is the record in the main data file.

It should be noted that if you have two records which are absolutely identical in every way, it will only ever be possible to delete the first of them—though how you would be able to distinguish which had been deleted is difficult to see.

2.8.20. Print card index

The final addition to the program is a module which allows you to output the whole of the card index to your printer. The module does not provide for printing out of an individual card, but this can be done anyway by executing a screen dump when the relevant card is displayed.

2.8.20.1. Lines 26000-26130

```
26000 ' *****
26010 ' Print card index
26020 ' *****
26030 e=SEEKRANK(1,0,0)
26040 WHILE e=0 OR e=101
26050   GET 1
26060   LPRINT SPACE$(9) STRING$(c+4,"_")
26070   FOR i=1 TO 1
26080     LPRINT prompt$(i) "  ! " buff$(i) " !";
26090     LPRINT CHR$(13) SPACE$(9) STRING$(c+4,"_")
26100   NEXT i
26110   e = SEEKNEXT(1,0)
26120 WEND
26130 RETURN
```

2.8.20.2. Commentary

Line 26030: The SEEKRANK command is used to find the first field in the file.

Lines 26040-26120: The cards are printed out, one by one, with a line separating each card, until the SEEKNEXT command indicates that the end of the file has been reached.

2.9. BUDGET

We now turn our attention to the most complex and difficult program you will encounter in the course of this book. Budget is a powerful and flexible financial aid which will enable you to plan finances over a period of twelve months. Perhaps the most important feature of the program is that it not only allows you to enter the actual income and payments you expect to be making over the period but also lets you set up a parallel set of figures based on 'what if' questions, eg, what if I were to enter into a commitment to pay a sum of X for six months, starting in May? Provided that you provide an accurate assessment of your income and spending, Budget will provide you with a picture of your finances month by month, both for the real figures and your hypothetical 'what if' figures. Properly used, it can provide some very interesting information on the financial status of a family or small business over a period—when the tight periods will be and when there might be a surplus.

Month	Jan	Feb	Mar	Apr	May	Jun	
Gas	+0	+0	+100	+0	+0	+100	+33
Rent	+200	+200	+200	+200	+200	+200	+200
*Birthday present	+0	+0	+100	+0	+0	+0	+0
test1	+500	+0	+0	+513	+0	+9	+143

Press a key to display analysis

Drive is A:

Fig. 2-13: Budget displaying part of a set of accounts

2.9.1. Data files

Budget is a very complex program and, though it is simple to use, the testing of the program as it is developed can be difficult. For that reason we shall enter the two modules which save and load data at the very beginning of the process of creating the program. This will enable you to save some sample data early on in the process of development and subsequently re-load it to conduct your tests, rather constantly re-enter data for testing from the keyboard.

2.9.1.1. Lines 20000-21220

```
20000 ' *****
20010 ' Load data
20020 ' *****
20030 PRINT cls$
20040 PRINT TAB(30) "LOAD DATA"
20050 PRINT FN locate$(1,12) ;
20060 LINE INPUT "Enter a file name ? ",file$
20070 OPEN "i",1,file$
20080 INPUT #1,mm,y
20090 FOR h = 0 TO 1
20100   INPUT #1,n(h)
20110   FOR i = 0 TO 11
20120     INPUT #1,c1(h,i),c2(h,i)
20130   NEXT i
20140   FOR i = 0 TO n(h)-1
20150     INPUT #1,pa$(h,i)
20160     FOR j = 0 TO 11
20170       INPUT #1,pa(h,i,j)
20180     NEXT j
20190   NEXT i
20200 NEXT h
20210 CLOSE 1
20220 h = 0
20230 GOSUB 13000
20240 GOSUB 14000
20250 RETURN
21000 ' *****
21010 ' Save data
21020 ' *****
21030 PRINT cls$
21040 PRINT TAB(30) "SAVE DATA"
21050 PRINT FN locate$(1,12) ;
```

```

21060 LINE INPUT "Enter a file name ? ",file$
21070 OPEN "o",1,file$
21080 WRITE #1,mm,y
21090 FOR h = 0 TO 1
21100   WRITE #1,n(h)
21110   FOR i = 0 TO 11
21120     WRITE #1,c1(h,i),c2(h,i)
21130   NEXT i
21140   FOR i = 0 TO n(h)-1
21150     WRITE #1,pa$(h,i)
21160     FOR j = 0 TO 11
21170       WRITE #1,pa(h,i,j)
21180     NEXT j
21190   NEXT i
21200 NEXT h
21210 CLOSE 1
21220 RETURN

```

2.9.2. Menu

A standard menu module with the addition of a few lines which allow you to tell the program whether you wish to work with the real or 'what if' data, which are entirely separated in the memory.

Home budget

Commands Available:

- 1) Display monthly analysis
- 2) Changes
- 3) New budget headings
- 4) Delete budget headings
- 5) Reset hypothetical figures
- 6) Reset month
- 7) Save data
- 8) Initialise
- 9) End program

Which do you require (1-9) ? █

Drive is A:

Fig. 2-14: The Budget main menu

2.9.2.1. Lines 11000-11390

```
11000 ' *****
11010 ' Menu
11020 ' *****
11030 z = 0
11040 WHILE z<>9
11050 PRINT cls$
11060 PRINT TAB(30) "Home budget"
11070 PRINT
11080 PRINT FN locate$(26,3) "Comamnds Available:"
11090 PRINT FN locate$(30,5) "1) Display monthly
analysis"
11100 PRINT FN locate$(30,7) "2) Changes"
11110 PRINT FN locate$(30,9) "3) New budget
headings"
11120 PRINT FN locate$(30,11) "4) Delete budget
headings"
11130 PRINT FN locate$(30,13) "5) Reset
hypothetical figures"
11140 PRINT FN locate$(30,15) "6) Reset month"
11150 PRINT FN locate$(30,17) "7) Save data"
11160 PRINT FN locate$(30,19) "8) Initilise"
11170 PRINT FN locate$(30,21) "9) End program"
11180 PRINT FN locate$(26,23) ;
11190 LINE INPUT "Which do you require (1-9) ? ",
t$
11200 z = VAL(t$)
11210 IF z>=5 AND z<9 THEN ON z-4 GOSUB 14000,
16000,21000,10000,19000 : z = 0
11220 WHILE z>0 AND z<5
11230 h = 0
11240 WHILE h<>1 AND h<>2
11250 PRINT cls$
11260 PRINT FN locate$(20,12) "1) Real data"
11270 PRINT FN locate$(20,14) "2) Hypothetical
data"
11280 PRINT FN locate$(20,16) ;
11290 LINE INPUT "Which do you require ? ",t$
11300 h = VAL(t$)
11310 WEND
11320 h = h-1
11330 ON z GOSUB 12000,18000,15000,19000
```

```

11340   z = 0
11350   WEND
11360   WEND
11370   PRINT cls$
11380   PRINT FN locate$(20,11) "Home budget
        terminated"
11390   END

```

2.9.3. Initialisation

A standard initialisation module.

2.9.3.1. Lines 10000-10530

```

10000 ' *****
10010 ' Initialise
10020 ' *****
10030 CLEAR
10040 cls$ = CHR$(27)+"E"+CHR$(27)+"H"
10050 rvson$ = CHR$(27)+"p"
10060 rvsoff$ = CHR$(27)+"q"
10070 DEF FN locate$(x,y) = CHR$(27)+"Y"+CHR$(32+
        y)+CHR$(32+x)
10080 DIM pa$(1,29),mo(1,29),pa(1,29,11),pt(1,11),
        bd(1,11)
10090 DIM c1(1,11),c2(1,11),ba(1,11),mo$(11)
10100 RESTORE 10000
10110 FOR i = 0 TO 11
10120   READ mo$(i)
10130   mo$(i) = LEFT$(mo$(i)+SPACE$(10),10)
10140 NEXT i
10150 WIDTH 255,255
10160 PRINT cls$
10170 PRINT TAB(30) "Home Budget"
10180 PRINT : PRINT
10190 LINE INPUT "Are you loading from disk (Y/N)
        ? ",t$
10200 IF LOWER$(t$)="y" THEN GOSUB 20000
10210 WHILE LOWER$(t$)<>"y"
10220   LINE INPUT "What is the number of the
        current month (1-12) ? ",t$
10230   mm = VAL(t$)-1
10240   y = mm+11

```

```

10250 GOSUB 17000
10260 GOSUB 15000
10270 GOSUB 14000
10280 t$ = "y"
10290 WEND
10500 DATA January,February,March
10510 DATA April,May,June
10520 DATA July,August,September
10530 DATA October,November,December

```

2.9.3.2. *Commentary*

Lines 10080-90: The main arrays used in the program are as follows:

PA\$: The names of the payment and income items—the two sides of the array (0 and 1) will refer to the real and ‘what if’ items.

PA: The amounts associated with each of the payments, broken down into 12 monthly figures.

MO: The average monthly payment for each payment heading.

PT: The total payments for each month.

BD: The balance of budgeted payments over actual payments.

C1: The figures for main income.

C2: The figures for supplementary income.

BA: The cash balance at the end of each month.

MO\$: The names of the months.

Lines 10100-10140: The names of the months, contained in the DATA lines beginning at 10500, are read into the array MO\$.

Lines 10190-10290: At this point the user has two choices: either to load a previously stored set of budget figures from disc, or to make an initial input of income and payment figures.

2.9.4. Income

This module accepts a set for 12 figures for main and supplementary income. The reason for the two different figures is that probably a majority of people have one main source of income on which they rely. In addition, however, they often also have the facility to add to that income on a regular or irregular basis.

2.9.4.1. Lines 17000-17200

```

17000 ' *****
17010 ' Income
17020 ' *****
17030 PRINT cls$
17040 PRINT "Input main income as follows:"
17050 FOR i = mm TO y
17060   i1 = i MOD 12
17070   PRINT mo$(i1) ":";
17080   LINE INPUT " ? ",t$
17090   c1(h,i1) = VAL(t$)
17100 NEXT i
17110 PRINT cls$
17120 PRINT "Other anticipated income:"
17130 FOR i = mm TO y
17140   i1 = i MOD 12
17150   PRINT mo$(i1) ":";
17160   LINE INPUT " ? ",t$
17170   c2(h,i1) = VAL(t$)
17180 NEXT i
17190 GOSUB 13000
17200 RETURN

```

2.9.4.2. Commentary

Line 17060: Data is stored in the arrays in the order January-December but the program is capable of starting its 12 month period at any point during the year. The variable I1 is used to translate the loop variable, which runs from the number of the current month to the current month plus 12, into a number in the range zero to 11.

Line 17090: It is worth noting, at this point, the way in which the variable H is used to determine which side of the arrays, the real or what-if, the figures are to be placed into. When called from the initialisation module the figures will always go into the real side of the arrays but later we shall add modules which will permit what-if figures to be input by altering the value of H.

2.9.5. Input of payments

This module accepts the relevant facts about the payments to be made over the year. For each payment there will be a payment name and 12 figures, one for each month of the year. One extra piece of information is whether the payment is to be budgeted. The budgeting facility calculates an average level of payments over a period of 12 months which would be necessary to cover a particular payment heading. When the budget for a particular month is drawn up, the program will *either* enter the average payment for the heading *or* the actual payment for the month if the item has been excluded from the budgeting process.

2.9.5.1. Lines 15000-15190

```
15000 ' *****
15010 ' Input of payments
15020 ' *****
15030 WHILE -1
15040 PRINT cls$
15050 PRINT TAB(30) "Input of bills"
15060 PRINT
15070 PRINT "Precede name of item with a '*' if
      you do not want it budgeted."
15080 PRINT "('ZZZ' to quit)"
15090 INPUT "Heading for bill: "; q$
15100 IF UPPER$(q$)="ZZZ" THEN GOSUB 13000 :
      RETURN
15110 n(h) = n(h)+1
15120 IF n(h)=30 THEN n(h)=29 : PRINT CHR$(7) "
      No more room" : FOR i = 0 TO 1000 : NEXT :
      GOSUB 13000 : RETURN
15130 pa$(h,n(h)-1)=q$ : PRINT "Payments under "
      q$ ": "
15140 FOR i = mm TO y
15150 i1 = i MOD 12
15160 PRINT mo$(i1) ": ";
15170 INPUT pa(h,n(h)-1,i1)
15180 NEXT i
15190 WEND
```

2.9.5.2. Commentary

Line 15070: If you wish to exclude an item from the budgeting process, the way to do it is to enter the payment name preceded by an asterisk, eg *INSURANCE.

Line 15080: The program will go on asking you for payments until you enter one with the name 'ZZZ'.

Line 15110: Once again the variable H is used to determine a difference between the real and what-if sides of the arrays. In the case of this line, the two element array N is used to record how many payments are recorded on each side of the arrays—the two figures may be quite different.

Lines 15140-15180: Each payment heading has twelve monthly figures associated with it, even if 11 of them are zero. Once again, the variable I1 is used to translate between the number of the month within the period being surveyed and the actual month within the January-December period.

2.9.6. Update budget

Though we do not yet have anything like the full range of facilities, we do have the ability to enter a set of working figures. Our ultimate aim is to create a table which can be displayed on screen and, though we cannot actually create the display itself, we can enter this module, which calculates the figures for it.

2.9.6.1. Lines 13000-13290

```

13000 ' *****
13010 ' Update budget
13020 ' *****
13030 t(h) = 0
13040 FOR i = 0 TO n(h)-1
13050   bu = 0
13060   IF LEFT$(pa$(h,i),1)="*" THEN t = 12 ELSE
      t = 0
13070   FOR j = t TO 11
13080     bu = bu+pa(h,i,j)
13090   NEXT j
13100   mo(h,i) = bu/12
13110   t(h) = t(h)+mo(h,i)
13120 NEXT i
13130 tt = 0
13140 cu = 0
13150 FOR i = mm TO y
13160   i1 = i MOD 12
13170   pt(h,i1) = 0
13180   FOR j = 0 TO n(h)-1

```

```
13190  pt(h,i1) = pt(h,i1)+pa(h,j,i1)
13200  NEXT j
13210  tt = tt+pt(h,i1)
13220  FOR j = 0 TO n(h)-1
13230  IF LEFT$(pa$(h,j),1)="*" THEN tt = tt-pa(
    h,j,i1)
13240  NEXT j
13250  bd(h,i1) = t(h)*(i-mm+1)-tt
13260  cu = cu+c1(h,i1)+c2(h,i1)-pt(h,i1)
13270  ba(h,i1) = cu
13280  NEXT i
13290  RETURN
```

2.9.6.2. *Commentary*

Lines 13040-13120: These two nested loops calculate the monthly payments for each item included in the budgeting facility (ie, those whose name do not begin with '*') and place the monthly budget figure in the array MO. To accomplish this, all that needs to be done for each payment is to add up all the figures under that heading over the period and then divide by 12.

Lines 13150: The beginning of a loop which will perform a series of operations on the figures for each month.

Lines 13170-13200: For each month, these lines extract the total of the actual payments to be made and store it in the corresponding element of the array PT.

Line 13210: In addition to the monthly totals, the total for the period so far is cumulated in the variable TT by adding the monthly totals to that variable.

Lines 13220-13240: This loop subtracts from the yearly total the figures for any payments which are not included in the budgeting facility. What will be left in the variable TT is the total payments so far over the 12 month period which are included in the budgeting facility.

Line 13250: We now have two sets of figures we can make use of to give us some useful information. In the array T is stored the average monthly figure for all payments which fall under the budgeting facility. In the variable TT is stored the total actual payments which have been made on budgeted items. Subtracting TT from the contents of T gives a figure for the balance of the recommended budget figure over actual payments. The importance of this is that it tells you whether any apparent surplus or deficit of cash is due to irregularities in spacing of payments. If you have a series of heavy payments at the end of the twelve month period then you will appear, in the eleventh month, to have a substantial

cash surplus. The figure for the balance between the budget and the actual payments will show up how much of this surplus is actually money which is intended for payments to be made later.

Line 13260-13270: The total cash surplus or deficit is calculated for each month by subtracting the total payments for each month from the figures for main and supplementary income. The figure is cumulated in the array BA, which will contain a record of the total cash surplus/deficit at any point during the year.

2.9.7. Display Figures

The display section of a substantial program is often most complex of all the modules. The reason for this is that while there may often be an underlying logic to the module, a great deal of space will be taken up with lines necessary to position items on the screen in ways that make the data comprehensible. In most cases those lines will have been arrived at by trial and error rather than applying any neat programming methods.

This module is no exception to that rule. Once it has been entered and you have seen the way in which the figures are placed on the screen, the whole thing will become a great deal clearer than any amount of the accompanying explanation can make it.

2.9.7.1. Lines 12000-12830

```

12000 ' *****
12010 ' Display worksheet
12020 ' *****
12030 m1 = -1 : WHILE m1<1 OR m1>12
12040 PRINT cls$
12050 PRINT TAB(30) "Worksheet"
12060 PRINT FN locate$(1,12) ;
12070 m1 = 0
12080 LINE INPUT "Number of month to start (1-12)
      : ",t$
12090 m1 = VAL(t$)
12100 WEND
12110 m1 = m1-1
12120 IF mm-m1-12*(m1>mm-1)<6 THEN m1 = mm-6-12*(
      mm<=6)

```



```
12130 q$ = "y"
12140 WHILE q$="y"
12150 PRINT cls$ SPC(14) rvson$ SPC(65) rvsoff$
12160 PRINT SPC(14) rvson$ " " rvsoff$ " Month ";
12170 FOR j = m1 TO m1+5
12180 PRINT rvson$ " " rvsoff$ " " LEFT$(mo$(j
MOD 12),3) " ";
12190 NEXT j
12200 PRINT rvson$ " " rvsoff$ " " rvson$ "
" rvsoff$
12210 PRINT rvson$ SPC(79) rvsoff$
12220 FOR i = 0 TO n(h)-1
12230 t = -1
12240 WHILE i=15 AND t
12250 t = 0
12260 PRINT : PRINT "Press a key to clear
screen and continue";
12270 WHILE INKEY$=""
12280 WEND
12290 PRINT FN locate$(0,3);
12300 FOR j = 1 TO 20
12310 PRINT SPACE$(79)
12320 NEXT j
12330 PRINT FN locate$(0,3);
12340 WEND
12350 PRINT rvson$ " " rvsoff$ " " LEFT$(pa$(h,
i)+SPACE$(20),20);
12360 FOR j = m1 TO m1+5
12370 PRINT rvson$ " " rvsoff$ " " USING "+####
";INT(pa$(h,i,j MOD 12)) ;
12380 PRINT " ";
12390 NEXT j
12400 PRINT rvson$ " " rvsoff$;
12410 PRINT " " USING "+####";INT(mo$(h,i));
12420 PRINT " " rvson$ " " rvsoff$
12430 NEXT i
12440 PRINT rvson$ SPC(79) rvsoff$
12450 PRINT : PRINT "Press a key to display
analysis" ;
12460 WHILE INKEY$=""
12470 WEND
12480 PRINT FN locate$(0,3) ;
12490 FOR i = 1 TO 20
```

```

12500 PRINT SPC(79)
12510 NEXT i
12520 RESTORE 12000
12530 PRINT FN locate$(0,3) ;
12540 FOR i = 1 TO 8
12550 READ a$
12560 PRINT rvson$ " " rvsoff$ a$
12570 PRINT rvson$ SPC(79) rvsoff$
12580 NEXT i
12590 t = 22
12600 FOR i = m1 TO m1+5
12610 i1 = i MOD 12
12620 PRINT FN locate$(t,3) rvson$ " " rvsoff$ "
" USING "+####";pt(h,i1)
12630 PRINT FN locate$(t,5) rvson$ " " rvsoff$ "
" USING "+####";t(h)
12640 PRINT FN locate$(t,7) rvson$ " " rvsoff$ "
" USING "+####";bd(h,i1)
12650 PRINT FN locate$(t,9) rvson$ " " rvsoff$ "
" USING "+####";c1(h,i1)
12660 PRINT FN locate$(t,11) rvson$ " " rvsoff$
" " USING "+####";c2(h,i1)
12670 PRINT FN locate$(t,13) rvson$ " " rvsoff$
" " USING "+####";c1(h,i1)+c2(h,i1)
12680 PRINT FN locate$(t,15) rvson$ " " rvsoff$
" " USING "+####";c1(h,i1)+c2(h,i1)-pt(h,i1)
12690 PRINT FN locate$(t,17) rvson$ " " rvsoff$
" " USING "+####";ba(h,i1)
12700 t = t+8
12710 NEXT i
12720 FOR i = 1 TO 8
12730 PRINT FN locate$(70,i*2+1) rvson$ " "
rvsoff$ SPC(7) rvson$ " " rvsoff$
12740 NEXT i
12750 PRINT FN locate$(1,20) ;
12760 INPUT "Do you wish to review the figures (
y/n) ",q$
12770 q$ = LOWER$(q$)
12780 WEND
12790 RETURN
12800 DATA Total,Budget,Budget Bal.
12810 DATA Main Income,Supp. Income
12820 DATA Total Income,Cash Balance
12830 DATA Cum. Balance

```

2.9.7.2. Commentary

Lines 12030-12110: Before the table of figures can be displayed, the program needs to know the point at which the display is to begin, since it can begin at any month within the 12 month period covered by the data. These lines accept a month number in the range one to 12, representing the number of the calendar month to begin the display (not the number of the month within the 12 month period to be covered). The figure input is reduced by one to take account of the fact that the arrays are numbered from zero, not one.

Line 12120: This line is somewhat less than comprehensible at first sight, but it serves a very simple purpose. What the formula in the first half of the line does is to determine whether the suggested start month is less than six months from the end of the 12 month period currently held in memory. The eventual display will be for a sequence of six months and there is no point in going to the trouble of making the program display less than this. Accordingly, if the suggested start point for the table would provide less than six months up to the end of the period, the start point will be moved back so that it provides a picture of the last six months of the period.

The strange calculation in brackets is what is known as a logical condition. M1, as we have already seen, is the number of the month at which you wish to start the display of figures. MM represents the number of the calendar month in which the current 12 month period begins. The usefulness of the logical condition is that “ $(M1 > MM-1)$ ” will be assessed as having a *value* according to whether the month number input is less than the actual number of the calendar month at which the current 12 month period starts. Any such logical condition will be regarded as nothing more less than the figure zero if the condition it expresses is false, ie, M1 is not greater than MM-1, or -1 if it is true.

For instance, you might be working with a period which begins in April and runs to March, so the value of MM would be three (the calendar month value of April minus one to take account of the fact that the arrays start from zero). If you told the program that you wished it to start from July (month five), then you would be faced with the ludicrous situation that the time until the end of the period will be calculated as from month five (July) up to month three (March), or *minus* two months. The truth is that the remainder of the period is actually July till the *following* March, which is month three *plus* 12.

What happens is that the expression $(M1 > MM-1)$ is translated by the machine as meaning the number zero if it is false, ie, if the start point is before the month MM-1 representing the end of the period, (eg, if the end of the period were month 8, and you wanted to start at month 2). The figure from the first half

of the line would then be $8-2-12*(0)$, or a sensible value of six months to the end of the period. If the start point was apparently *after* the end period, however, such as with a start point of eight and an end point of two, then $(M1 > MM-1)$ would be translated by the machine as having the value -1 and the calculation would become $2-8-12*(-1)$, or $2-8+12$, giving the result that there are six months to the end of the period.

The second part of the line uses the same technique to set the start point of the display to at least six months before the end of the current 12 month period, even if that means the month number of the start point is actually greater than the month number of the end point (ie, it falls at the end of the previous year).

This is a long explanation for one single line but the technique embodied here is a very important one that can be used to substantially shorten programs where calculations rely heavily on logical conditions.

Line 12140-12210: Having determined where the display of figures is to start, the program now turns to creating the screen display. These lines create the outline of the table using spaces printed in reverse video and then print the first three letters of each month name in the specified six month period along the top of the screen. The use of MOD in determining the month to be printed ensures that if the program arrives at December, the next month will be January.

Line 12220: This loop will be executed as many times as there are payments on the current side of the array, as represented by the value of H.

Line 12230-12340: If there are more than 15 payments to be handled, they cannot all be displayed on the same screen. In that case this loop will come into play to provide a pause.

Lines 12350-12390: The payment name is printed at the left hand side of the screen up to a maximum of 20 characters, followed by a reversed space. The payments associated with that heading are then placed onto the screen, again separated by reversed spaces.

Lines 12400-12420: At the extreme right hand edge of the table is printed the monthly budget figure for the particular item, or zero if it is excluded from the budgeting facility.

Lines 12450-12510: Once all the payments and their budget figures have been displayed, the program will pause. The next stage, when the user presses a key, is to display the analysis based on the payments. To do this, the main part of the screen will be cleared, leaving only the month names at the top.

Lines 12520-12580: In this second half of the table we shall be working with preset headings on which our analysis is based. These lines read the headings from the data at the end of the module and places them down the left hand edge of the screen.

Lines 12600-12710: This loop, which will be executed for each of the six months being displayed, places the relevant figures for each heading in the column for the appropriate month. The contents of the different arrays used in creating the table are described in the commentary on previous modules, especially 'Update Budget'.

Line 12760: The greater part of this module is taking place inside one large loop, which will be executed if the variable Q\$ contains the letter Y (or y). If, having examined the analysis, the user wishes to have another look at the payments for the same period, all that is necessary is to answer Y at this point and the whole table display will begin again, but without the need to specify a start month.

2.9.8. Find budget head

In the next module we shall give ourselves the ability to change the values associated with items. Before we do so, however, we shall enter this short module which searches for a particular payment heading and returns its position or informs the user that the name of the payment in question has not been found.

2.9.8.1. Lines 22000-22190

```
22000 ' *****
22010 ' Find budget head
22020 ' *****
22030 PRINT : PRINT
22040 PRINT prompt$ ;
22050 LINE INPUT " ? ",q$
22060 q$ = UPPER$(q$)
22070 pp = -1
22080 FOR i = 0 TO n(h)-1
```

```

22090 IF q$=UPPER$(pa$(h,i)) THEN pp = i
22100 NEXT i
22110 found = pp>=0
22120 t = found
22130 WHILE NOT t
22140 PRINT CHR$(7) "Item " q$ " was not found"
22150 FOR i = 0 TO 1000
22160 NEXT i
22170 t = -1
22180 WEND
22190 RETURN

```

2.9.8.2. Commentary

Lines 22130-22180: If the item is not found within the array, an error message is displayed.

2.9.9. Changes

If a change is required to an item which has already been entered, then this module will allow you to specify whether the item is a payment heading, or main or supplementary income. The relevant figures are then displayed, month by month for the current 12 month period. For each figure you will have the choice of either entering a new value or a single '*', which will leave the current value unchanged.

2.9.9.1. Lines 18000-18450

```

18000 ' *****
18010 ' Changes
18020 ' *****
18030 PRINT cls$
18040 PRINT TAB(30) "Changes"
18050 PRINT FN locate$(26,5) "Commands available:"
18060 PRINT FN locate$(30,9) "1) Change budget
      head"
18070 PRINT FN locate$(30,11) "2) Change main
      income"
18080 PRINT FN locate$(30,13) "3) Change
      additional income"
18090 PRINT FN locate$(26,17) ;
18100 LINE INPUT "Which do you require ? ",t$
18110 qq = VAL(t$)

```

```
18120 ON qq GOSUB 18150,18310,18310
18130 GOSUB 13000
18140 RETURN
18150 prompt$ = "Name of budget head to be
changed"
18160 GOSUB 22000
18170 WHILE found
18180 found = 0
18190 PRINT cls$
18200 PRINT pa$(h,pp)
18210 PRINT FN locate$(0,4) "Enter new amount or
'*' to leave"
18220 PRINT
18230 FOR i = mm TO y
18240 i1 = i MOD 12
18250 PRINT mo$(i1) pa(h,pp,i1) ": " ;
18260 LINE INPUT q$
18270 IF q$<>"*" THEN pa(h,pp,i1)=VAL(q$)
18280 NEXT i
18290 WEND
18300 RETURN
18310 PRINT cls$
18320 IF qq=2 THEN PRINT TAB(26) "Main" ; ELSE
PRINT TAB(26) "Additional";
18330 PRINT " income ('*' to leave unchanged)"
18340 FOR i = mm TO y
18350 i1 = i MOD 12
18360 PRINT mo$(i1) ;
18370 IF qq=2 THEN PRINT c1(h,i1); ELSE PRINT c2
h,i1);
18380 PRINT ": " ;
18390 LINE INPUT q$
18400 WHILE q$<>"*"
18410 IF qq=2 THEN c1(h,i1) = VAL(q$) ELSE c2(h,
i1) = VAL(q$)
18420 q$ = "*"
18430 WEND
18440 NEXT i
18450 RETURN
```

2.9.9.2. *Commentary*

Line 18130: After any change is made, the module at 13000 which updates the

budget figures is automatically called to ensure that the figures are accurate for the current contents of the arrays.

Lines 18150-18300: This section is called if the user specifies that the item to be changed is a payment heading. Provided that the module at 22000 confirms that the item is present in the array, the name is printed out and then each value, together with the month under which it registers. The user is requested to make a new input for the value and this input is stored in place of the existing value provided that it does not consist of a single '*'.

Lines 18310-18450: These lines deal with changes to both main and supplementary income. The procedure is the same as for the previous section except that there is no need to look for a payment name and the program will decide whether the contents of the main or supplementary income arrays are to be displayed and modified.

2.9.10. Delete budget head

This module allows you to delete a payment heading from the current list, together with its associated figures.

2.9.10.1. Lines 19000-19160

```

19000 ' *****
19010 ' Delete budget head
19020 ' *****
19030 prompt$ = "Name of budget head to delete"
19040 GOSUB 22000
19050 WHILE found
19060   found = 0
19070   n(h) = n(h)-1
19080   FOR i = pp TO n(h)-1
19090     pa$(h,i) = pa$(h,i+1)
19100     FOR j = 0 TO 11
19110       pa(h,i,j) = pa(h,i+1,j)
19120     NEXT j
19130   NEXT i
19140   GOSUB 13000
19150 WEND
19160 RETURN

```


2.9.10.2. *Commentary*

Lines 19080-19130: These two loops collapse the arrays to overwrite the item which has been specified for deletion. The process has to be done in two parts, one to overwrite the name of the item and then another to overwrite the 12 payment values associated with it. These are all the deletions that need to be done, since every other trace of the item will be erased when the figures are updated.

Line 19140: Having erased an item, the budget figures are now inaccurate and the update module must be called.

2.9.11. **Update month**

Budget is rendered more useful than it might otherwise be by the fact that it does not have to stick with a single period of 12 months. As time passes, this module will allow you to inform the program that instead of commencing with, for instance, April, you now wish to move on and have a start month of June to reflect the passing of time. The real purpose of the current module is to extract from you, when the start month is changed, the figures for income and payments for the new month or months which have been added to the end of the original 12 month period. Months which fall before the new start month will be forgotten, so if you wish to keep a continuous historical record of the results of the program over a period of more than 12 months you will have to take a copy of the original file before changing the start month.

2.9.11.1. *Lines 16000-16370*

```
16000 ' *****
16010 ' Update month
16020 ' *****
16030 m2 = -1
16040 WHILE m2<0 OR m2>11
16050   PRINT cls$
16060   PRINT TAB(26) "Update month"
16070   PRINT
16080   LINE INPUT "Number of new current month ? "
16090   ,t$
16090   m2 = VAL(t$)-1
16100 WEND
16110 WHILE m2 MOD 12<>mm
16120   IF m2<mm THEN m2 = m2+12
```

```

16130  FOR i = mm TO m2
16140    i1 = i MOD 12
16150    PRINT c1s$
16160    PRINT TAB(26) "Update month"
16170    PRINT
16180    PRINT "Input full amounts for next " mo$(
      i1) ":"
16190    PRINT
16200    FOR j = 0 TO n(0)-1
16210      PRINT pa$(0,j) " (" pa(0,j,i1) "): ";
16220      LINE INPUT t$
16230      pa(0,j,i1) = VAL(t$)
16240    NEXT j
16250    PRINT
16260    LINE INPUT "Main income: ",t$
16270    c1(0,i1) = VAL(t$)
16280    LINE INPUT "Additional income: ",t$
16290    c2(0,i1) = VAL(t$)
16300  NEXT i
16310  mm = m2 MOD 12
16320  y = mm+11
16330  h = 0
16340  GOSUB 13000
16350  GOSUB 14000
16360  WEND
16370  RETURN

```

2.9.11.2. *Commentary*

Line 16110: This loop will only be carried out if the month input is different from the current start month.

Line 16120: The new start month is adjusted if it is smaller than the current month and therefore refers to a month in the subsequent year.

Lines 16130-16300: This loop is carried out once for each month from the one following the current last month to the new last month. For each month the user is requested to supply a figure for each of the payment headings, and for main/supplementary income.

Line 16310-16320: The calendar number of the new start month is stored in MM and the new last month in Y.

Lines 16330-16350: Having changed the basic figures, the budget must be recalculated. In addition, though we have not yet entered the module to perform the task, since the whole basis of the budget has been changed, the what-if side of the arrays is also reset to conform to the new set of real figures. Once again, if you wish to keep a record of previous what-if figures, you must take a copy of the disc file before making the changes.

2.9.12. Set up what-if arrays

You can at any point in the course of the program use the facilities to insert, delete or modify data on the what-if side of the arrays to create a separate set of figures to examine hypothetical situations. The most common use of the what-if facilities, however, is to make small changes to the current real situation and examine their effect. Rather than force you to re-enter all the data which you have built up on the real side of the arrays, this module, which can be called from the main menu and is automatically called if you change the start month, copies the entire contents of the real side of the arrays into the what-if side. At the conclusion of this module, the two sides of the arrays are exactly identical in their contents, though either can of course be subsequently changed without affecting the other.

2.9.12.1. Lines 14000-14170

```
14000 ' *****
14010 ' Set up the shadow arrays
14020 ' *****
14030 t(1) = t(0)
14040 FOR i=0 TO n(0)-1
14050   pa$(1,i) = pa$(0,i)
14060   mo(1,i) = mo(0,i)
14070   FOR j = 0 TO 11
14080     pa(1,i,j) = pa(0,i,j)
14090   NEXT j
14100 NEXT i
14110 FOR i = 0 TO 11
14120   bd(1,i) = bd(0,i)
14130   cl(1,i) = cl(0,i)
14140   ba(1,i) = ba(0,i)
14150 NEXT i
14160 n(1) = n(0)
14170 RETURN
```

2.9.13. Testing the program

In order to check the table, it is necessary to explain what the various headings in the analysis part are intended to display:

TOTAL: The total of all payments to be made during a month.

BUDGET: The same figure for each month, representing the average sum which will have to be set aside over a year to pay for all the items included in the budgeting facility. Note that the budget figure will not necessarily correspond at all to what has been paid out up to the current month, since payments may be bunched at a particular point during the year.

BUDGET BAL.: The difference between the amount set aside in average BUDGET allocations since the beginning of the 12 months and what has actually been paid out on items included in the budgeting facility.

MAIN/SUPPLEMENTARY INCOME: The actual figures you entered for each month against these headings.

TOTAL INCOME: The sum of the previous two headings for each month.

CASH BALANCE: The difference between income and outgoings during the month.

CUM. BALANCE: The difference between income and outgoings since the beginning of the 12 month period in question.

There will be small discrepancies in the figures displayed since only integer figures are being used in the table, whereas the calculations are being performed on the full figures. For an annual payment of 47 pounds, for instance, the monthly budget figure will be displayed as three pounds. This will not in any way affect the accuracy of the overall monthly budget figure for all the budgeted items. For ten items at 47 pounds, the overall monthly budget allocation would be 39 pounds, not 30 pounds. In other words, the overall recommended budget figure will never be more than a few pence out.

Month	Jan	Feb	Mar	Apr	May	Jun	
Total	+700	+200	+400	+713	+200	+309	
Budget	+377	+377	+377	+377	+377	+377	
Budget Bal.	-323	-146	-68	-404	-227	-159	
Main Income	+1000	+1000	+1000	+1000	+1000	+1000	
Supp. Income	+50	+200	+50	+200	+50	+200	
Total Income	+1050	+1200	+1050	+1200	+1050	+1200	
Cash Balance	+350	+1000	+650	+487	+850	+891	
Cum. Balance	+350	+1350	+2000	+2487	+3337	+4228	

Do you wish to review the figures (y/n) ☐

Drive is A:

Fig.2-15: The Budget analysis screen

2.10. TRIVIA

The last program in this section represents both a useful tool and a little light relief. The essence of the program is that it is a multiple choice question generator, ie, it sets questions and then provides four possible answers from which you must choose.

The program can be used seriously as an enjoyable and addictive way of refreshing your memory on a particular topic or, as the name implies, as a light hearted question master on questions of no great consequence. The range of uses is limited only by your own willingness to provide the questions and answers which are its raw material.

Though the application is different from anything which has gone before, it is interesting to note that in structure Trivia is just another data handling program and its methods reflect closely what has gone before in programs like Unifile and NNumber.

ENTER for next question or '[' to quit ?

Fig. 2-16: A reward for a correct answer

Like several of the previous programs, Trivia is a chameleon, capable of changing its appearance according to the data it is given to work on. In this initialisation module, the user has the opportunity either to load a previously created set of questions or to set up a new set.

```

10000 ' *****
10010 ' Setup system
10020 ' *****
10030 cls$=CHR$(27)+"E"+CHR$(27)+"H"
10040 rvson$=CHR$(27)+"p"
10050 rvsoff$=CHR$(27)+"q"
10060 DEF FN locate$(x,y)=CHR$(27)+"Y"+CHR$(32+y)+
CHR$(32+x)
10070 WHILE NOT in
10080 PRINT cls$

```

```
10090 PRINT TAB(36) "TRIVIA"
10100 PRINT TAB(36) "====="
10110 in = -1
10120 DIM err.type$(2),na$(1),q(4),a$(1,499),d$(
9)
10130 DIM d(1,9)
10140 PRINT : LINE INPUT "Are you loading from
disc (y/n) ? ",q$
10150 IF UPPER$(q$)="Y" THEN GOSUB 16000
10160 WHILE UPPER$(q$)<>"Y"
10170 PRINT cls$
10180 PRINT TAB(30) "Test Structure"
10190 PRINT TAB(30) "====="
10200 PRINT FN locate$(1,6) ; : LINE INPUT "
Name for question ? ",na$(1)
10210 PRINT : LINE INPUT " Name for answer ? ",
na$(0)
10220 PRINT : LINE INPUT " Are these correct (y/
n) ? ",q$
10230 d$(0) = "Untyped"
10240 ty = 1
10250 IF UPPER$(q$)="Y" THEN GOSUB 12000
10260 WEND
10270 err.type$(0) = "No more room for types"
10280 err.type$(1) = "No more room for items"
10290 err.type$(2) = "You must have at least
five questions entered"
10300 in = -1
10310 WEND
```

2.10.1.2. *Commentary*

Line 10120: The more important arrays used during the course of the program are as follows:

ERR.TYPE\$: Used to contain a set of three error messages which the program is capable of generating.

NA\$: A two element array containing the names given to the questions and answers, eg, EVENT and DATE.

Q: An array which will be used to store a list of five possible answers when a question is set.

A\$: The main array containing questions and answers.

D: An array containing pointers to the start positions of groups of similar questions/answers in the main array.

D\$: An array containing the names of groups of similar questions in the main array.

Lines 10200-10220: Each set of questions you enter can have a unique name for the questions and answers. If you were setting a French test, for instance, you might call the questions 'French word' and the five possible answers 'English word'.

Lines 10230-10240: Questions can be grouped into a number of 'types', in order to make the tests slightly harder by ensuring that possible answers are drawn only from similar types. In the case of the hypothetical French test, each question and answer might be given a label specifying that it was a noun, adjective and so on. When the program is first initialised there is only one type, called 'Untyped'. It is up to the user whether to include other types or stick with one, though once entered it is not possible to change the type of a question without deleting and re-entering it.

2.10.2. Error

This short module is called whenever one of the three error messages is to be displayed. All that needs to be done is to set the value of the variable `ERR.TYPE` before calling the module with `GOSUB`.

2.10.2.1. Lines 21000-21060

```
21000 ' *****
21010 ' Error
21020 ' *****
21030 PRINT CHR$(7) "***** " err.type$(err.type) "
      *****"
21040 FOR i = 0 TO 10000
21050 NEXT i
21060 RETURN
```

2.10.3. New types

As already mentioned, the program provides the opportunity to divide up the questions and answers out of which it creates a test into different types. This

module, which accepts new type names, is normally called when the program is first initialised but it can also be accessed from the main program menu subsequently.

2.10.3.1. Lines 12000-12240

```
12000 ' *****
12010 ' New types
12020 ' *****
12030 WHILE -1
12040 PRINT cls$
12050 PRINT TAB(30) "New types"
12060 PRINT TAB(30) "======"
12070 PRINT : PRINT TAB(20) "Enter '[' to quit"
12080 PRINT : PRINT TAB(20) "Types entered so
      far"
12090 PRINT
12100 IF ty=1 THEN PRINT TAB(26) "None"
12110 FOR i = 1 TO ty-1
12120   PRINT TAB(26) i "- " d$(i)
12130 NEXT i
12140 PRINT
12150 IF ty=10 THEN err.type = 0 : GOSUB 21000 :
      RETURN
12160 PRINT : PRINT TAB(20) ; : LINE INPUT "New
      type ? ",nt$
12170 IF nt$="[" THEN RETURN
12180 PRINT : PRINT : LINE INPUT "Is this
      correct (y/n) ? ",z$
12190 WHILE UPPER$(z$)="Y"
12200   z$ = "n"
12210   d$(ty) = nt$
12220   ty = ty+1
12230 WEND
12240 WEND
```

2.10.3.2. Commentary

Lines 12160-12230: A new type name is input and stored in the array D\$ at the point indicated by the variable TY, which records the number of different types which has been entered. The program is set up to accept 10 different types (including 'untyped').

2.10.4. Menu

A standard menu module.

```

TRIVIA
=====

Command available:

1) Enter new items
2) Search/delete
3) Enter new types
4) Generate questions
5) Display or reset scores
6) Save data
7) Load data/Initialise
8) End

Enter command required: █

```

Drive is A:

Fig. 2-17: The Trivia main menu

2.10.4.1. Lines 11000-11230

```

11000 ' *****
11010 ' Menu
11020 ' *****
11030 command = -1
11040 WHILE command<>8
11050 PRINT cls$
11060 PRINT TAB(30) "TRIVIA"
11070 PRINT TAB(30) "======"
11080 PRINT FN locate$(20,6) "Command available:"
11090 PRINT FN locate$(25,9) "1) Enter new items"
11100 PRINT TAB(26) "2) Search/delete"
11110 PRINT TAB(26) "3) Enter new types"
11120 PRINT TAB(26) "4) Generate questions"
11130 PRINT TAB(26) "5) Display or reset scores"

```

```
11140 PRINT TAB(26) "6) Save data"
11150 PRINT TAB(26) "7) Load data/Initialise"
11160 PRINT TAB(26) "8) End"
11170 PRINT FN locate$(19,20) ; : LINE INPUT "
      Enter command required: ",c$
11180 command = VAL(c$)
11190 IF command=7 THEN RUN
11200 ON command GOSUB 13000,18000,12000,19000,
      20000,17000
11210 WEND
11220 PRINT cls$
11230 END
```

2.10.5. Entry of new items

This module accepts the input of new questions and answers, together with any type label which the user wishes to attach to them.

2.10.5.1. Lines 13000-13530

```
13000 ' *****
13010 ' Entry of new items
13020 ' *****
13030 t1$ = ""
13040 t2$ = ""
13050 WHILE t1$<>"[" AND t2$<>"["
13060 exit = 0
13070 PRINT cls$
13080 PRINT TAB(30) "New items"
13090 PRINT TAB(30) "=====
13100 PRINT : PRINT "Enter '[' to quit"
13110 PRINT
13120 IF it>=500 THEN err.type = 1 : GOSUB 21000
      : RETURN
13130 PRINT na$(1) ":" ; : LINE INPUT t2$
13140 WHILE t2$<>"[" AND NOT exit
13150 PRINT : PRINT na$(0) ":" ; : LINE INPUT
      t1$
13160 WHILE t1$<>"[" AND NOT exit
13170 IF ty=1 THEN t = 0 ELSE t = -1
13180 WHILE t<0
13190 PRINT cls$
13200 PRINT TAB(30) "New items"
```

```

13210     PRINT TAB(30) "======"
13220     PRINT : PRINT TAB(20) "Types available:"
        : PRINT
13230     FOR i = 1 TO ty
13240         PRINT TAB(26) i "- " d$(i-1)
13250     NEXT i
13260     PRINT : PRINT TAB(20) ; : LINE INPUT "
Enter type required ? ",ty$
13270     t = VAL(ty$)-1
13280     IF t>ty THEN t=-1
13290     WEND
13300     PRINT cls$
13310     PRINT TAB(30) "New items"
13320     PRINT TAB(30) "======"
13330     PRINT : PRINT na$(1) ": " t2$
13340     PRINT : PRINT na$(0) ": " t1$
13350     PRINT : PRINT "Type: " d$(t)
13360     PRINT : LINE INPUT "Is this correct (y/n)
? ",q$
13370     WHILE UPPER$(q$)="Y"
13380         d(0,t) = d(0,t)+1
13390         t1$ = CHR$(48+t)+t1$
13400         GOSUB 14000
13410         GOSUB 15000
13420         q$ = "n"
13430     WEND
13440     exit = -1
13450     WEND
13460     WEND
13470     WEND
13480     su = 0
13490     FOR i = 0 TO 9
13500         d(1,i) = su
13510         su = su+d(0,i)
13520     NEXT i
13530     RETURN

```

2.10.5.2. Commentary

Line 13050: The new question and answer will be accepted into the temporary variables T1\$ and T2\$. To terminate the module at any point the user only has to enter the '[' character for either.

Lines 13180-13290: This section, which displays the names of the types currently entered, is only displayed if the user has actually entered new type names over and above the default untyped label.

Line 13380: The record of the number of questions in the particular type group, held in the array D, is updated.

Lines 13390: When questions are stored in the main array, the type they belong to is attached to the front of the question in the form of a character representing the type number. In the ASCII character set used by Basic, the numbers 0 to 9 start with zero at position 48. Adding character 48 + 2 to the front of a string simply adds the number '2'. The reason for adding a character to the front in this way is that when the question is inserted into the main array in strict alphabetical order, all the questions of the same type will automatically be placed together.

Lines 13400-13410: Calls to the modules, not yet entered, which find the correct position in the main array for the new question and insert it.

Lines 13480-13520: We have already added one to the record in the zero side of the D array of the number of items in the group to which the new question belongs. One further thing is necessary, however, since the other side of the D array contains a record of where, in the main array, the first item of every group is to be found. To do this, the program simply scans through the record of the size of each group, cumulating them — ie, if the first two groups contain 10 and 10 respectively, then the third group must start after 20 entries.

2.10.6. Binary search

A standard binary search which works on the basis of the question entered. Since the type number is attached to the front of the question, questions will automatically be sorted by type.

2.10.6.1. Lines 14000-14110

```
14000 ' *****
14010 ' Binary search
14020 ' *****
14030 IF it=0 THEN ss=0 : RETURN
14040 po = INT(LOG(it)/LOG(2)) : ss = 2^po-1
14050 FOR i = po TO 0 STEP -1
14060 IF a$(0,ss)<t1$ THEN ss = ss+2^i ELSE IF a$
      (0,ss)>t1$ THEN ss=ss-2^i
```

```

14070 IF ss<0 THEN ss=0
14080 IF ss>=it THEN ss = it-1
14090 NEXT i
14100 IF a$(0,ss)<t1$ THEN ss = ss+1
14110 RETURN

```

2.10.7. Insert item

A standard insert module which takes the position arrived at by the previous module and shifts the data in the array up to provide a space at that point.

2.10.7.1. Lines 15000-15110

```

15000 ' *****
15010 ' Insert item
15020 ' *****
15030 FOR i = it-1 TO ss STEP -1
15040   FOR j = 0 TO 1
15050     a$(j,i+1) = a$(j,i)
15060   NEXT j
15070 NEXT i
15080 a$(0,ss) = t1$
15090 a$(1,ss) = t2$
15100 it = it+1
15110 RETURN

```

2.10.8. Data files

Two standard modules which allow you store a set of questions on disc and subsequently reload them.

2.10.8.1. Lines 16000-17170

```

16000 ' *****
16010 ' Load data
16020 ' *****
16030 PRINT cls$
16040 PRINT TAB(30) "Load data"
16050 PRINT TAB(30) "=====
16060 PRINT : LINE INPUT "Enter filename ? ",
      filename$
16070 OPEN "i",1,filename$

```

```
16080 INPUT #1,ty,it
16090 FOR i = 0 TO ty-1
16100 INPUT #1,d$(i),d(0,i),d(1,i)
16110 NEXT i
16120 FOR i = 0 TO it-1
16130 INPUT #1,a$(0,i),a$(1,i)
16140 NEXT i
16150 INPUT #1,na$(0),na$(1)
16160 CLOSE 1
16170 RETURN

17000 ' *****
17010 ' Save data
17020 ' *****
17030 PRINT cls$
17040 PRINT TAB(30) "Save data"
17050 PRINT TAB(30) "======"
17060 PRINT : LINE INPUT "Enter filename ? ",
      filename$
17070 OPEN "o",1,filename$
17080 WRITE #1,ty,it
17090 FOR i = 0 TO ty-1
17100 WRITE #1,d$(i),d(0,i),d(1,i)
17110 NEXT i
17120 FOR i = 0 TO it-1
17130 WRITE #1,a$(0,i),a$(1,i)
17140 NEXT i
17150 WRITE #1,na$(0),na$(1)
17160 CLOSE 1
17170 RETURN
```

2.10.9. User search/delete

As with the previous data handling programs, this one provides you with the ability to page through the items in the main array and to delete items. The module is a relatively simple combined one since you are not given the option to amend current entries, only to examine and delete them. If you do wish to make changes, you must delete an item and then re-enter it.

2.10.9.1. Lines 18000-18370

```
18000 ' *****
18010 ' User search/delete
18020 ' *****
```

```

18030 ss = 0
18040 WHILE -1
18050   PRINT cls$
18060   PRINT TAB(30) "Search"
18070   PRINT TAB(30) "====="
18080   IF ss>=it THEN ss = it-1
18090   IF ss<0 THEN ss = 0
18100   PRINT : PRINT TAB(20) "Number of items ="
      it
18110   PRINT : PRINT : PRINT TAB(20) "Commands
      available:"
18120   PRINT : PRINT TAB(26) "ENTER for next item"
18130   PRINT TAB(26) "Pos/neg number to move
      pointer"
18140   PRINT TAB(26) "'D' to delete item"
18150   PRINT TAB(26) "'I' to quit"
18160   PRINT : PRINT TAB(20) "Entry -" ss+1
18170   PRINT : PRINT TAB(26) na$(1) ": " a$(1,ss)
18180   PRINT TAB(26) na$(0) ": " MID$(a$(0,ss),2)
18190   PRINT TAB(26) "Type: " d$(ASC(a$(0,ss))-48)
18200   PRINT : PRINT TAB(20) ; : LINE INPUT "
      Which command do you require ? ",q$
18210   IF q$="" THEN q$="1"
18220   WHILE UPPER$(q$)="D"
18230     t = ASC(a$(0,ss))-48
18240     d(0,t) = d(0,t)-1
18250     FOR i = t TO ty-1
18260       d(1,t) = d(1,t)-1
18270     NEXT i
18280     FOR i = ss TO it-1
18290       a$(0,i) = a$(0,i+1)
18300       a$(1,i) = a$(1,i+1)
18310     NEXT i
18320     it = it-1
18330     q$ = ""
18340   WEND
18350   ss = VAL(q$)+ss
18360   IF q$="I" THEN RETURN
18370 WEND

```


2.10.9.2. *Commentary*

Lines 18230-18270: When an item is deleted it is not simply a matter of removing it from the main array. There is also the need to amend the array D, which contains a record of how many items there are in each type group and where each group starts in the main array.

2.10.10. **Questions**

This is the only really original module in the whole of the program. Everything else you have entered is to be found, in one form or other, in other programs in the book. This one module is completely unique because its job is to set the tests which are the sole claim to fame of Trivia. The main elements of the module are that it chooses a random question from the mass of questions in the main array and then selects five possible answers to that question by mixing up the correct answer with the answers to four other questions.

Where the four other answers will come from will depend upon how the user has set up the test. There are two options, one to take the four wrong answers from anywhere in the array and a second to take answers only from within the same group as the random question. The difference between the two ways of setting up the test is that if answers are drawn from the whole file, they are more prone to look unlikely as answers, and the test will be accordingly easier. If you set up your type groups sensibly, so that the questions within each type are on similar topics, then limiting the possible answers to the same group will provide you with incorrect answers which will be far more difficult to distinguish from the correct one.

Apart from selecting the random question and answers, the module also accepts the users choice of answer, informs the user whether that answer is correct or not and updates the score for the current test.

2.10.10.1. *Lines 19000-19580*

```
19000 ' *****
19010 ' Questions
19020 ' *****
19030 PRINT cls$
19040 PRINT TAB(30) "Questions"
19050 PRINT TAB(30) "=====
19060 PRINT
19070 IF d(1,ty-1)<5 THEN err.type=2 : GOSUB
      21000 : RETURN
```

```

19080 LINE INPUT "Do you wish answers to be drawn
      only from the same question type (y/n) ? ",q$
19090 qu = UPPER$(q$)<>"Y"
19100 WHILE -1
19110 PRINT cls$
19120 PRINT TAB(30) "Questions"
19130 PRINT TAB(30) "=====
19140 PRINT
19150 q1 = INT(RND(1)*it-1)
19160 t = ASC(a$(0,q1))-48
19170 IF d(0,t)<5 OR qu THEN p1=0 : p2=it-1 ELSE
      p1=d(1,t) : p2=d(0,t)
19180 FOR i = 0 TO 3
19190   ok = 0
19200   WHILE NOT ok
19210     pp = p1+INT(RND(1)*p2)
19220     ok = pp<>q1
19230     FOR j = 0 TO i
19240       ok = ok AND pp<>q(j)
19250     NEXT j
19260   WEND
19270   q(i) = pp
19280 NEXT i
19290 q2 = INT(RND(1)*5)
19300 q(4) = q(q2)
19310 q(q2) = q1
19320 PRINT : PRINT : PRINT na$(1) ": " a$(1,q1)
19330 PRINT : PRINT na$(0) ": "
19340 PRINT
19350 FOR i = 0 TO 4
19360   PRINT i+1 " ) " MID$(a$(0,q(i)),2)
19370 NEXT i
19380 qt = qt+1
19390 PRINT : LINE INPUT "Which do you think is
      the number of the correct answer (1-5) ? ",n$
19400 WHILE VAL(n$)-1=q2
19410   PRINT cls$
19420   FOR i = 0 TO 19
19430     PRINT FN locate$(60-i,i) "Correct"
19440   NEXT i
19450   FOR i = 1 TO 6
19460     PRINT FN locate$(i*3+19,i+14) "Correct"
19470   NEXT i

```

```
19480 rr = rr+1
19490 q2 = -1
19500 WEND
19510 WHILE q2>=0
19520 PRINT : PRINT TAB(20) "Wrong"
19530 PRINT : PRINT "The correct answer was: "
      MID$(a$(0,q1),2)
19540 q2 = -1
19550 WEND
19560 PRINT : PRINT : LINE INPUT "ENTER for next
      question or 'I' to quit ? ",q$
19570 IF q$="I" THEN RETURN
19580 WEND
```

2.10.10.2. *Commentary*

Line 19080-19090: The user's decision as to whether possible answers will be drawn only from the same question type is stored in the variable QU. The value of QU will reflect the value of the logical expression `UPPER$(Q$)<>"Y"`. The use of logical conditions as values was discussed in the course of the Budget program. The effect of line 19090 is that QU will equal -1 if the answer to the prompt is 'Y' (or 'y') and zero if it is not.

Line 19150: This line places a value into the variable Q1, which is the position in the array of the question which will be asked.

Line 19160: The variable T is set equal to the value of the first character of the question which, as we have seen previously, is the number of the type attached to the question.

Line 19170: This line sets the upper and lower limit in the array from which the possible answers will be chosen. If there are less than five answers in the same group as the chosen question, or if QU is equal to -1, then the upper and lower limits will be the end and beginning of the array, respectively.

Lines 19180-19280: These lines choose the four wrong answers. The procedure is that a random answer is chosen in the range indicated by the limits P1 and P2. The position of this answer is then compared with the position of the correct answer, to make sure that they are not the same. The position is then compared with those of incorrect answers which have been previously chosen, to ensure that no answers are duplicated. If either of two tests is failed, the value of the variable OK will be set to zero to indicate the problem, and the process of choosing will be carried out again.

It is worth remembering that it is the positions of the answers which are compared, not the answers themselves. If two questions have the same answer, then it is quite possible that the same answer will appear twice and only one will be accepted as right. In addition, if you are choosing questions from type groups which have only five or six answers in each, the program will slow down somewhat since it will frequently choose an answer which has already been selected and have to go through the process again.

When an incorrect answer is chosen and found not to be the same as either the correct answer or any previously chosen correct answer, it is placed into one element of the array Q. The four incorrect answers are originally placed into the first four elements (0-3) of Q.

Lines 19290-19310: These lines take one of the incorrect answers chosen and place it into element four of Q. Once this has been done, the correct answer is placed into Q to replace it. This does not mean that the correct answer is never in position four, since the program is perfectly happy to copy the non-existent answer in position four back into its own position and then replace it with the correct answer.

Lines 19320-19390: The selected question is displayed on the screen, together with the five possible answers and the user is invited to input the number of the correct answer. The variable QT records the total number of questions asked in this session and is increased by one.

Lines 19400-19500: If the answer given is the number of the correct answer in the array Q, the screen displays a large tick made up of the word 'correct'. Line 19480 increases the value of RR, the variable which records the number of correct answers.

Lines 19510-19550: If the wrong answer is input, the user is informed of the fact and the correct answer is displayed.

Lines 19560-19570: After each question, the user has the option to press ENTER for another question or input '[' to terminate the current quiz session.

2.10.11. Score

This short module displays the current score. Calculating the score is not quite as simple as it might seem at first sight, since it would clearly make no sense to take the total of right answers as a proportion of the overall total. As there are

only five answers provided on the screen at any one time, random button punching would give a score of 20%. Somehow, we must eliminate this element of luck from the score. The simplest way to do it is to say that the wrong answers registered represent only 4/5th of the questions asked where the user did not know the answer—ie, there is another fifth where the button pressed just happened to be the right one. This extra allocation of wrong answers is subtracted from the right answers before the percentage mark is given. If there are no wrong answers, a perfect score will be given, whereas if 80% of answers are wrong the score will be zero. If you answer every question wrongly you will end up with a score of minus 20, indicating, quite rightly, that your performance is even worse than would have been achieved by random button pushing.

The module also allows the score to be reset to zero so that a new quiz can be started. If the score is not reset it cumulates every time the quiz module is used.

2.10.11.1. *Lines 20000-20130*

```
20000 ' *****
20010 ' Score
20020 ' *****
20030 PRINT cls$
20040 PRINT TAB(30) "Score"
20050 PRINT TAB(30) "====="
20060 IF qt=0 THEN RETURN
20070 PRINT : PRINT TAB(20) "Total number of
      questions:" qt
20080 PRINT : PRINT TAB(20) "Number of correct
      answers:" rr
20090 PRINT : PRINT TAB(20) "Score is:" INT((rr-(
      qt-rr)/4)/qt*100) "%"
20100 PRINT : PRINT : PRINT : PRINT
20110 PRINT TAB(20) ; : LINE INPUT "Do you wish
      to zero your score (y/n) ? ",q$
20120 IF UPPER$(q$)="Y" THEN qt = 0 : rr = 0
20130 RETURN
```

CHAPTER 3

GSX

The PCW range of computers comes complete with a special graphics package which goes under the name of GSX. The full name of the system is Graphics Extension System and it has been developed over a number of years by Digital Research Inc, the designers of CP/M.

The object of GSX is to provide a common means for a variety of different computers to display high quality graphics on screen and/or a printer of some kind. Programmers who wish to employ graphics in their program need only to build the correct GSX commands into their work in order to automatically make the most of the facilities on a machine. The hard work of actually making the machine produce the graphics is done by the designers of GSX in adapting the package to different computers.

GSX comes in a variety of different shapes and sizes. On expensive up-market computers a full GSX system has a quite astonishing range of facilities, far more than will normally be used. Other machines may use the same system but be more limited in the facilities which are available. Compared to the full GSX system, the version provided free with the PCW range is in some ways a fairly limited tool, but only limited when compared to the full potential of GSX. There are many owners of home and personal computers which proudly boast their graphics capabilities but which would fall far short of what can be achieved on a PCW.

The main problem with the GSX package on the PCW is not a limitation on the graphic effects it can achieve but on its use in practice. Because GSX is available, PCW owners have access to a wide range of programs which create displays based on GSX commands. What the average PCW owner cannot do is to make use of GSX in their own programs, because the BASIC language supplied is not capable, unaided, of communicating in the special way that GSX requires.

In the course of this chapter, readers of this book will find that limitation removed. Using the special techniques described, you will be able to write simple programs which call upon all of the facilities available under the PCW

version of GSX, with some spectacular results. Before going on to the programs which embody the GSX techniques, however, we need to take a look at GSX and the unique way in which it accepts commands to create graphics.

3.1. GSX in abstract

If we disregard the actual output of the system, what we are left with at the core of GSX is a complex piece of software which defines for itself a graphics screen consisting of 32767*32767 individual points—a resolution far in excess of anything which can be achieved on the 640*192 pixel screen of the PCW. Onto its imaginary screen, GSX is capable of placing a bewildering variety of items. It can draw lines of various styles and thicknesses, create rectangles, circles, arcs and irregular polygons. Shapes can be empty or they can be filled with a wide range of patterns. Text can be placed into a design at positions defined down to a single pixel, with the individual characters scaled by any factor, printed at any angle across the screen. Items on the screen can be copied or moved from one place to another, again with pixel precision and colours can be set to an almost infinite range of shades. In accordance with the needs of many modern programs, the system is capable of accepting information from a mouse, a small device used to move a pointer round the screen and perform actions independent of the keyboard.

Not all of these functions are available in the version of GSX supplied with the PCW, although most are. Just because the facilities are built into GSX, however, does not mean that you are going to be able to use all of them. To understand why that is, it is necessary to take a quick look at the concept of devices.

3.2. GSX and devices

If you recall the discussion of logical and physical devices in the chapter on CP/M, then the best way to think of GSX is as a logical device—it is to the logical GSX device that all the intended graphics output is sent. However, a GSX which is capable of creating a rather fine but entirely theoretical display is hardly of much use to anyone. What is required of a graphics package is that it should produce a display somewhere. To accomplish this, GSX makes use of what are known as virtual devices. This term applies to some physical devices which are fairly obvious—they include the screen of your monitor and the printer. Other devices are hardly physical, yet they are treated in the same way, which is why they are called virtual devices. Less obvious virtual devices

include special files which GSX can create on disk to store the contents of graphic designs for subsequent use.

In each case, whether it is outputting to the screen or some other more exotic device, GSX has to tailor its instructions to the potential of the device. There is no point, for instance, in sending complex colour information to a monochrome monitor or a black and white printer. There is no point in sending details of a 32767*32767 pixel design to a screen with only 640*190 positions. In fact it would seem at first sight that there is not much point in GSX doing anything, since there are almost no devices on the market capable of accepting in full the detail, colour and so forth that it can create.

The solution to this problem is what is known as a device driver, a small program specially designed to accept instructions from GSX and to translate them into a form which makes sense to a particular device. The device driver for the monitor, for instance, has the task of translating display positions which are defined by GSX co-ordinates into pixel positions on the screen.

Every device which is to be used by GSX must have a device driver, since GSX itself never tailors its output to match a particular device. The device driver, which is written specially to work with GSX, must (or rather *should*) be capable of accepting any GSX instruction and either converting it into a form compatible with the device or informing GSX that the particular action cannot be carried out.

In the case of impossible instructions, they will still be output by GSX but the user will be passed a message that the particular action cannot be carried out, though no reason will be given. When an instruction is not carried out, it may not be the fault of the device itself, it may simply be that the device driver has not been designed to make the maximum use of the facilities that the device offers.

It follows from all this that the effects which can be created by GSX will depend on both the device and its driver. The same GSX package may be capable of producing a colour display on a monitor, a black and white printout on a printer and so on. In the version of the GSX system supplied with the PCW you will find that the printer driver is actually more capable than the screen driver so that, for instance, fill patterns can be produced on the printer but not on the screen. The last program in this chapter represents a tool which will allow you to ask GSX exactly what it can do with a particular device driver. For the purposes of quick reference, the capabilities of the GSX package supplied at the time of publication are shown in the table at 3.4.

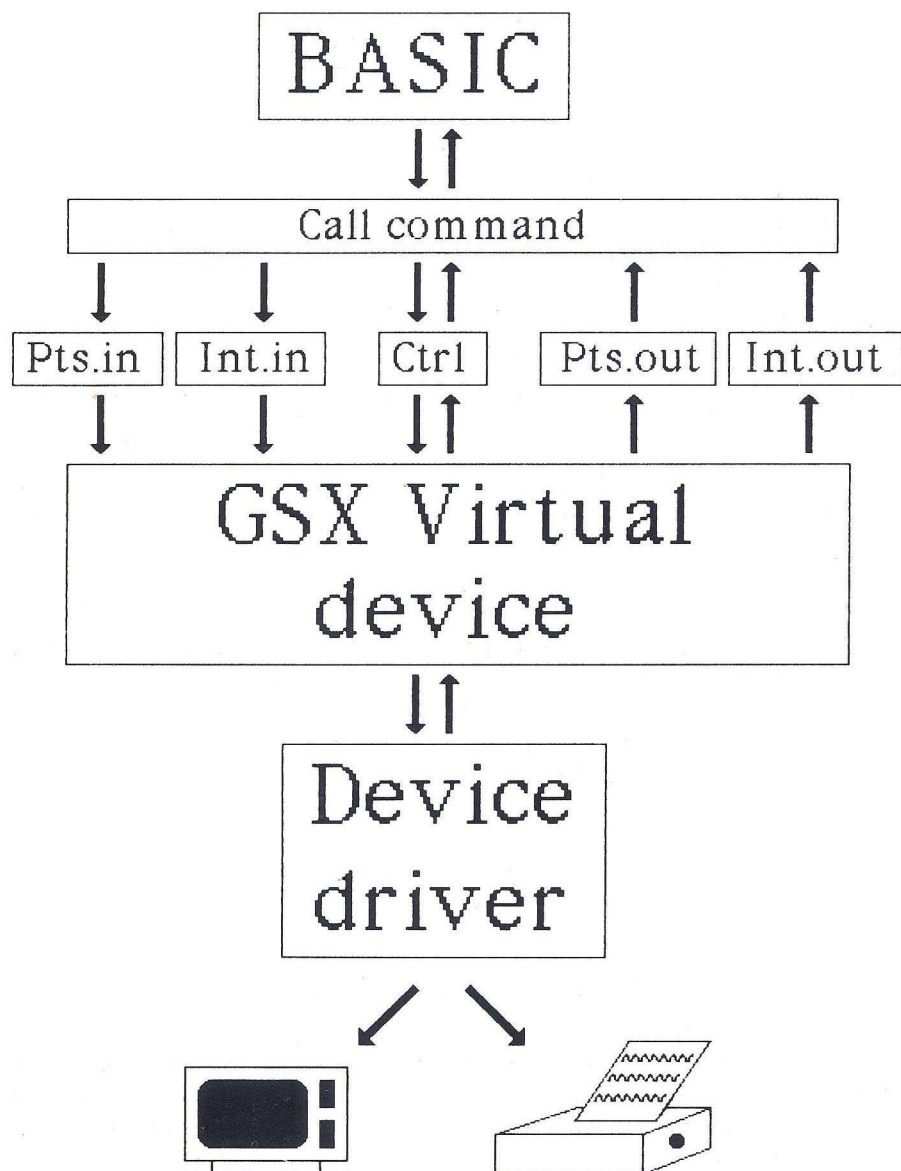


Fig. 3-1: GSX viewed as a logical device

3.2.1. Device drivers and the Assign.Sys file

The device drivers which are used by GSX are entirely different from those used by the rest of the system, since they need to be specially written to understand and interpret GSX commands. Because its device drivers are different, GSX cannot simply ask CP/M to choose the correct driver—it needs its own separate list, which is contained in a special text file called ASSIGN.SYS. The contents of the basic ASSIGN.SYS file delivered with GSX is as follows:

```
21 @:DDFXHR8
22 @:DDFXLR8
11 @:DDHP7470
01 @:DDSCREEN
```

The file is simple to understand, and to change if necessary, once a few points are noted:

1) Every device used by GSX has a unique two digit number. The numbers must fall into the range 01-10 for a monitor, 11-20 for a plotter and 21-30 for a printer:

a) It follows from this that DDFXHR8 and DDFXLR8 must be device drivers for printers. In fact they are both drivers for a printer which uses the same command set as the Epson FX range, which the PCW printer does. The reason that there are two different drivers is that one is high resolution for high-quality output and the other low resolution for faster output of draft material.

b) DDHP7470 is a driver for a Hewlett Packard plotter—a special kind of printer which draws with pens. A variety of plotters with a wide range of prices use the same command set as the Hewlett Packard machine.

c) DDSCREEN is just what the name implies, a driver for the screen.

2) The '@:.' part of the name signifies that when GSX wants to access the device driver it will be found on the default disc drive. Those who wish to place their device drivers on another drive will need to specify where it will be found—GSX will not search for it.

3) The name of the device driver omits the file terminator. The convention is that GSX device drivers have the terminator '.PRL'. You can use a driver with

another terminator provided that you include the terminator in the name recorded in the ASSIGN.SYS file.

4) The device drivers can be listed in any order except that *the largest file must be placed first*, regardless of the device it refers to or its number. The reason for this is that when GSX is initialised for use it sets aside an area of memory equivalent to the size of the first device driver in the list, into which the contents of device drivers are loaded when they are used. The size of this area of memory, once it is set up, cannot be changed. Accordingly, unless the biggest file is placed first, there will not be enough memory to use that device driver if it is ever needed.

This information is not really necessary unless you intend to add new devices to your PCW system. In that case, your first step must be to ensure that your dealer has demonstrated to you that the device is capable of working with the basic PCW. If you wish to use the new device for GSX output, however, you must also ensure that there is a suitable GSX compatible device driver *and* you will have to place the name of the device driver into the ASSIGN.SYS file using a text editor and give it an appropriate unused device number.

3.3. Using GSX

Making GSX work for you is really a very simple matter provided that you know how GSX receives commands and passes information back to the rest of the system—and you know how to send those commands. The commands themselves are listed in the table at 3.4 but if you want go in detail into the GSX system you would be well advised to purchase (through a computer dealer) a copy of the full GSX manual from Digital Research Inc. In the remainder of this section we shall give a brief outline of the way to set up GSX commands and how to pass them to GSX from Basic. Practical examples of the commands are given in the three programs which constitute the rest of the chapter.

3.3.1. Setting up GSX commands

All commands to GSX are issued in the form of numbers placed into a series of arrays in memory. By convention the arrays are named CTRL, PTSIN, INTIN, PTSOUT and INTOUT. There is no particular significance to these names except that they conform to the descriptions of the GSX commands given in the full GSX manual. In the programs later in this chapter we have made the names slightly more readable by renaming PTSIN as PTS.IN and so on.

The purpose of the different arrays is as follows:

>> CTRL—The control array. This array is used to tell GSX which of its many functions it is to carry out. In addition to a value specifying the command itself, some other information such as how many items are to be found in the data arrays may have to be included. The commands and further information necessary are included in the table at 3.4. In Basic, the array will normally be declared as CTRL%(6). This will provide space for six, not seven values as you might expect, since GSX always works from position one in an array, not position zero.

>> PTS.IN—The points co-ordinates array. Much of the work of GSX consists of placing items on the screen at specified positions or of drawing lines between points. In order to do this, GSX needs to be told what the relevant horizontal and vertical co-ordinates of the positions are and these co-ordinates are placed into the PTS.IN array in pairs. What will actually be done with the co-ordinates will depend on the command being carried out, but where PTS.IN is used, the control array must always contain a value specifying to GSX how many pairs of co-ordinates are being supplied. The size of the PTS.IN array need only be as large as the number of co-ordinates but since it will normally be used a number of times in the course of a program it is wise to set it up so that it will exceed your likely needs. In the programs which follow, the array is normally declared as PTS.IN%(2,75), giving the potential to specify 75 pairs of co-ordinates at one time.

>> INT.IN—The input parameter array. Not all of the values which are supplied to GSX are co-ordinates; some are simply integer values. For these, a third array is necessary and it declared in the programs which follow as INT.IN%(80), allowing 80 integer values to be supplied.

>> PTS.OUT—The output point co-ordinates array. Some commands to GSX require that the system receive back information in the form of a set of co-ordinates. When this happens the co-ordinates will be found in the PTS.OUT array. In the programs which follow the array is declared as PTS.OUT%(2,75).

>> INT.OUT—The output parameters array. As with input, single integer values can be received back from GSX as well as pairs of co-ordinates. These will be found in INT.OUT, which we have declared as INT.OUT%(45)

It is impossible to generalise too much about how the arrays are used in conveying individual commands, since each command will require different information to be sent to GSX. What *can* be said with certainty is that the

‘opcode’ for the command—the number representing a command in GSX—will always be placed into element one of the CTRL array. When you come across a GSX command in the following programs, the number placed into the first position of CTRL can always be used to translate the command into English by referring to the table at 3.4. If you wish to issue particular commands, the table will tell you the relevant opcode and what other information must be placed where in the arrays.

3.3.2. Calling GSX from Basic

NOTE: This section provides a simplified technical explanation of the techniques used to set up communication between Basic programs and GSX. The information contained in this section is not necessary for the use of GSX provided that you use the GSX modules from the demonstration programs later in the chapter.

As we have already noted, the main problem with the GSX package supplied with the PCW is that while it can provide graphics for specially designed GSX programs, it is not capable of being used directly from Basic. The reason for this is that GSX requires some information which Basic is not normally capable of providing and also that GSX has to be called using a tiny piece of machine code, the language understood by the PCW’s Z80 central processor chip. We cannot really go into a vast treatise here on machine code techniques so the description of what actually happens is relatively sketchy—the important point is that it works.

In the previous section we saw that GSX uses arrays to pass commands and information. Unfortunately it does not use arrays in the same sense that Basic does. There is no way in which GSX can simply be told ‘use INT.IN(1)’ since GSX has no way of knowing what INT.IN is. All that GSX expects is that when it is called up it will be given the start points of five areas of memory which represent, in order, the start points of the information we have described above as CTRL, INT.IN, PTS.IN, INT.OUT and PTS.OUT. To obtain this information it will expect to be supplied with the address in memory of a block containing five addresses which are the start points of the arrays.

In normal Basic, the start addresses of arrays are not stored in this way. To collect together addresses we have to make use of the CALL command. You will notice that in the programs later in this chapter the ‘Call GSX’ module contains a line beginning:

CALL a (d%,d%,ctrl%(1)...

For the moment we shall ignore the 'a' at the beginning and concentrate on the second part of the line.

When the CALL command is issued, one of the things it does, if required, is to place into the registers (or storage locations of the Z80 chip at the heart of the PCW) the addresses in memory of a number of variables specified by the user. The first two addresses go into the DE and HL registers. In the line in question we are not interested in those registers and we merely supply CALL with the name of a dummy variable, D%.

When CALL has dealt with the first two variables mentioned, it changes its way of working. Since there are only a limited number of registers available in the Z80, it would be foolish to go on filling them with the addresses of variables—many of the registers are needed for other purposes anyway. To get over this problem CALL looks at the rest of the list variables and stores away the address of each, in order, in an area of free memory. It then places the address of this area into the BC register of the Z80. Earlier we pointed out that what GSX needs to be given is the address of a block of memory containing the start addresses of the five arrays it uses. The CALL command has put us halfway there by providing us with the five addresses and where they can be found in memory.

We can now turn back to the first part of the CALL statement, 'CALL a'. What this does is to run the machine code program which begins at the address in memory represented by the variable A. In this case, the machine code program to be run is a series of 10 values stored in an array called CODE% when the Basic program is first run. The variable A is the address of the first value in that array, obtained using the VARPTR function. For those who are interested in machine code programming, the content of the machine code program in assembly language is as follows:

```
PUSH BC
PULL DE
LD C, #73
JP #0005
```

The first two lines take the contents of the BC register and transfer them to the DE register, which is where GSX expects to find the location of the five addresses of the arrays it will work with. The second two lines ask CP/M to execute 'command 115' (73 in the hexadecimal numbering system). In actual fact there is no command 115 in normal CP/M, it is a special command which

GSX adds to CP/M when it is installed using the instructions in the next section. Once installed, command 115 simply tells CP/M to run GSX.

Simply put then, the flow of events is that the CALL statement places the addresses of the five arrays into memory, then the machine code program runs GSX and tells it where to find the arrays.

It should be stressed that you do not need to understand any of this in order to use GSX. Provided that you place the initialisation module shown in the programs which follow at the beginning of your own programs and copy the GSX control modules, you will be able to issue commands to GSX without having to worry about how they are carried out.

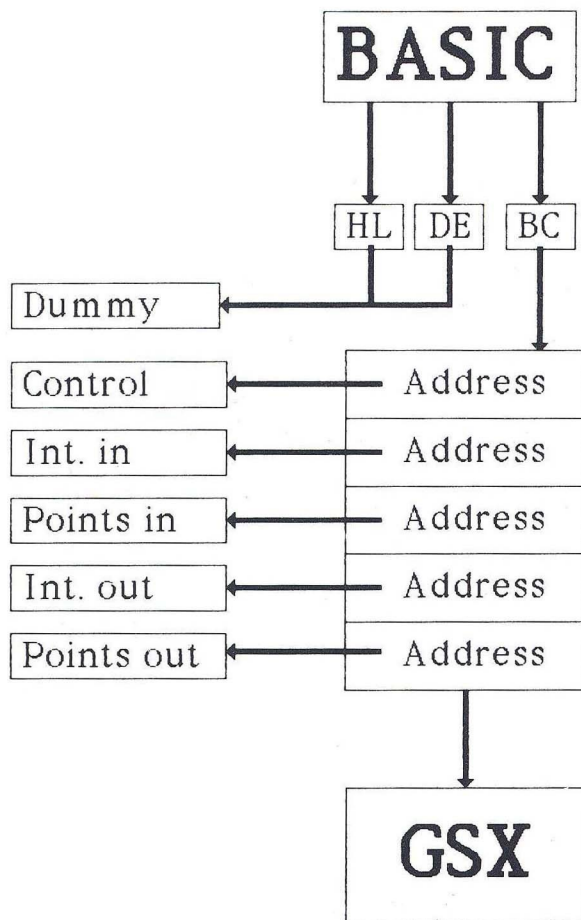


Fig. 3-2: A schematic view of GSX/Basic communication

3.3.3. Installing GSX

Having explained how instructions are issued to GSX and how the GSX system is called up, there remains only to describe how to set up a disk so that you can make use of what you have learned.

What you need is a disk with at least 100K of free space, onto which you can copy the following files:

DDSCREEN.PRL
DDFXHR8.PRL
DDFXLR8.PRL
GSX.SYS
GENGRAF.COM
BASIC.COM, renamed as GBASIC.COM (standing for Graphics Basic)
ASSIGN.SYS

If you have any other devices attached to your system which you wish GSX to make use of, you must also copy their device driver files and add their names to the ASSIGN.SYS file as described earlier in the chapter.

Having copied the files, type at the command line:

GENGRAF GBASIC

What this command does is to 'attach' GSX.SYS to the program which runs the Basic language, so that whenever GBasic is run, the GSX system will also be installed in memory. You can remove it again by typing the command for a second time. If you have no further need for the GENGRAF program it can now be deleted from the disk.

Attaching GSX increases the size of GBASIC.COM itself and in GBasic must make room for the device driver at the top of the list in the ASSIGN.SYS file. What this means is that you have far less free memory when using GBasic than in using a version of Basic to which GSX has not been attached, normally some 13K less. For that reason, many of the programs listed in the Basic chapter, which make use of large arrays in memory to store data, will not run under GBasic, so ensure that you keep copies of both Basic and GBasic to perform their different functions.

Once having created the new GBasic, whenever you type 'GBASIC' at the CP/M command line you will see something like the following:

A)gbasic

GSX-80 1.1 01 Oct 83 Serial No 5000-1232-654321
Copyright (C) 1983
Digital Research, Inc. All Rights Reserved

Mallard-80 BASIC with Jetsam Version 1.29
(c) Copyright 1984 Locomotive Software Ltd
All rights reserved

17255 free bytes

Ok

■

Drive is A:

Fig. 3-3: The GSX and Basic start-up messages

Like the normal Basic, GBasic can be installed on a disk so that it is run whenever the system is re-set with that disk in the drive. For the techniques of achieving this, see Chapter 1 on CP/M.

3.4. A table of GSX commands and usage

The object of this table is to provide a summary of the main functions available under the version of GSX supplied with the PCW range. For each command the table shows the name of the command, the opcode which must be placed into element one of the CTRL array and any further data to be placed into the the arrays INT.IN and PTS.IN. For examples of the use of the commands, refer to the programs which make up the remainder of the chapter. The GSX__TEST program at the end of the chapter will also provide you with a complete listing of the styles of line, fill, colour, etc, that the GSX package can provide.

Note: In GSX terminology, the device drivers with which the system works are known as workstations. In the table and the programs which follow, this terminology is adhered to, but the word 'workstation' can be translated as 'device driver' without confusion.

3.4.1. Open GSX workstation

Function: Notifies GSX that output is going to be sent to a particular device.

Note: Details of the line styles, etc, which can be set up as defaults using this command are included with the specific commands referring to the particular feature.

Input data:

CTRL%(1): 1

CTRL%(2): 0

CTRL%(4): 10

INT.IN%(1) Device number—See ASSIGN.SYS

INT.IN%(2) Initial polyline style.

INT.IN%(3) Initial polyline colour.

INT.IN%(4) Initial marker type.

INT.IN%(5) Initial polymarker colour.

INT.IN%(6) Initial text font.

INT.IN%(7) Initial text colour.

INT.IN%(8) Initial fill style.

INT.IN%(9) Initial fill index.

INT.IN%(10) Initial fill colour.

Output data:

CTRL%(3): 6

CTRL%(5): 45

PTS.OUT% array: Definition of this driver—See GSX test program.

INT.OUT% array: Definition of this driver—See GSX test program.

3.4.2. Close GSX workstation

Function: Closes down the previously opened GSX workstation

Input data:

CTRL%(1): 2

CTRL%(2): 0

Output data:

CTRL%(3): 0

3.4.3. Clear GSX Workstation

Function: Flushes out any buffers in memory currently being used to store information which needs to be sent to GSX.

Input data:

CTRL%0(1): 3

CTRL%0(2): 0

Output data:

CTRL%0(3): 0

3.4.4. Update GSX workstation

Function: Ensures that all data contained in memory buffers is sent to the current workstation and that any tasks pending for that workstation are carried out.

Input data:

CTRL%0(1): 4

CTRL%0(2): 0

Output data:

CTRL%0(3): 0

3.4.5. Place graphics cursor

Function: Positions a removable marker on the screen.

Input data:

CTRL%0(1): 5

CTRL%0(2): 2

CTRL%0(6): 18

PTS.IN%0(1,1): X position at which to place graphics cursor.

PTS.IN%0(2,1): Y position at which to place the graphics cursor.

Output data:

CTRL%0(3): 0

3.4.6. Remove last graphics cursor drawn

Function: Removes the last marker, without destroying other screen contents. If two markers have been placed on the screen, only one can be removed.

Input data:

CTRL%(1): 5

CTRL%(2): 0

CTRL%(6): 19

Output data:

CTRL%(3): 0

3.4.7. Draw a polyline

Function: Takes the points input and draws a series of lines between them.

Input data:

CTRL%(1): 6

CTRL%(2): Number of points to be drawn.

PTS.IN% array: Location of points to draw lines between.

Output data:

CTRL%(3): 0

3.4.8. Draw a polymarker

Function: Takes the points input and places the specified marker at each of them. Used for marking important points on line graphs, etc.

Input data:

CTRL%(1): 7

CTRL%(2): Number of points to be drawn.

PTS.IN% array: Location of points at which to draw markers.

Output data:

CTRL%(3): 0

3.4.9. Draw text

Function: Places text onto the screen. The style and size of text must have been determined by previous commands.

Input data:

CTRL%₀(1): 8

CTRL%₀(2): 1

CTRL%₀(4): Number of characters in text.

INT.IN% array: ASCII values of text, 1 per element of INT.IN%.

PTS.IN%₀(1,1): X position at which to draw the text.

PTS.IN%₀(2,1): Y position at which to draw the text.

Output data:

CTRL%₀(3): 0

3.4.10. Draw a filled polygon

Function: Takes the points input and draws a polygon based upon the corners represented by those points, then fills the polygon with the fill pattern specified with the 'fill' and 'style' commands.

Input data:

CTRL%₀(1): 9

CTRL%₀(2): Number of points in polygon outline.

PTS.IN% array: Coordinates for polygon outline.

Output data:

CTRL%₀(3): 0

3.4.11. Generalized drawing primitive (GDP)—Bar

Function: Draws a rectangular shape based on the position of two opposite corners specified.

Input data:

CTRL%₀(1): 11

CTRL%₀(2): 2

CTRL%₀(6): 1

PTS.IN%₀(1,1): X Coordinate of top left corner of bar.

PTS.IN%₀(2,1): Y Coordinate of top left corner of bar.

PTS.IN%(1,2): X Coordinate of bottom right corner of bar.

PTS.IN%(2,2): Y Coordinate of bottom right corner of bar.

Output data:

CTRL%(3): 0

3.4.12. Set the height of a character

Function: Sets the size of the characters to be printed using the 'draw text' command.

Input data:

CTRL%(1): 12

CTRL%(2): 1

PTS.IN%(1,1): 0

PTS.IN%(2,1): Requested character height.

Output data:

CTRL%(3): 2

PTS.OUT%(1,1) Character width.

PTS.OUT%(2,1) Character height.

PTS.OUT%(1,2) Character cell width.

PTS.OUT%(2,2) Character cell height.

3.4.13. Set polyline style

Function: Sets the type of line to be used in drawing a polyline.

Input data:

CTRL%(1): 15

CTRL%(2): 0

CTRL%(4): 1

INT.IN%(1): Polyline style 1-5

1 = Solid line. '————'

2 = Dash line. '— — — —'

3 = Dotted line. '.....'

4 = Dash dot line. '— . — . — . — .'

5 = Long dash line. '— — —'

Output data:

CTRL%(3): 0

CTRL%(5): 1

INT.OUT%(1): Polyline style actually selected.

3.4.14. Select polymarker type

Function: Selects the type of polymarker to be used with the draw polymarker command.

Input data:

CTRL%(1): 18

CTRL%(2): 0

CTRL%(4): 1

INT.IN%(1): Polymarker type required (1-5)

1 = '.' (Dot)

2 = '+' (Plus)

3 = '*' (Star)

4 = 'O' (Circle)

5 = 'X' (Cross)

Output data:

CTRL%(3): 0

CTRL%(5): 1

INT.OUT%(1): Polymarker type selected.

3.4.15. Set fill style

Function: Sets the type of filling to be used with a polygon.

Input data:

CTRL%(1): 23

CTRL%(2): 0

CTRL%(4): 1

INT.IN%(1): Requested fill style (0-3)

0 = Hollow (No fill)

1 = Solid

2 = Grey scale

3 = Hatch

Output data:

CTRL%(3): 0

CTRL%(5): 1

INT.OUT%(1): Fill style selected.

3.4.16. Set fill index**Function:** Sets the detail of the fill pattern chosen using the previous command.**Input data:**

CTRL%(1): 24

CTRL%(2): 0

CTRL%(4): 1

INT.IN%(1): Fill index required (1-6)

STYLE = HATCH	GREY SCALE
1 = Vertical lines	or white
2 = Horizontal lines	or very light grey
3 = +45 degree lines	or light grey
4 = -45 degree lines	or medium grey
5 = cross lines	or dark grey
6 = diamond pattern	or black

Output data:

CTRL%(3): 0

CTRL%(5): 1

INT.OUT%(1): Fill index actually selected.

3.4.17. Set writing mode**Function:** Sets the mode in which output will be placed on the screen. In the current version of GSX for the PCW, only the replace mode is available.**Input data:**

CTRL%(1): 32

CTRL%(2): 0

CTRL%(4): 1

INT.IN%(1): Writing mode required (1-4)

1 = REPLACE

2 = TRANSPARENT

3 = INVERT

4 = ERASE

Output data:

CTRL%(3): 0

CTRL%(5): 1

INT.OUT%(1): Writing mode selected.

3.5. Pie Chart

Having looked at some of the background to the way the GSX system works , we turn to two programs which demonstrate its use and a third program which allows you interrogate your GSX system to discover its capabilities. The first program in this section produces a standard pie chart display of a set of data. In the course of the program it will become clear that once you have lost your fear of the initially rather impenetrable information which GSX needs to be provided, the system is capable of far more sophisticated work than the Logo programs which are reproduced later.

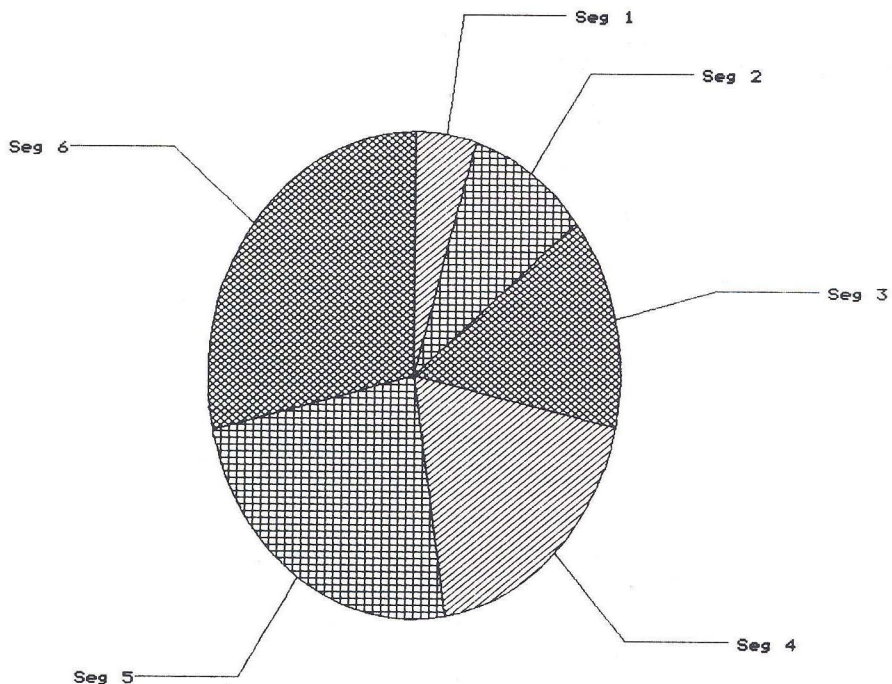


Fig. 3-4: A typical pie chart on printer

3.5.1. Initialise GSX system

This is the module which sets up the arrays which will be used to pass information to GSX and the small machine code routine to activate the system.

3.5.1.1. Lines 10000-10170

```

10000 ' *****
10010 ' Initialise GSX system
10020 ' *****
10030 OPTION BASE 1
10040 DIM code%(10),ctrl%(6),int.in%(80),pts.in%(
    2,75)
10050 DIM int.out%(45),pts.out%(2,75)
10060 d% = 0
10070 a = 0
10080 RESTORE 10170
10090 FOR i = 1 TO 7
10100   READ t
10110   POKE VARPTR(code%(1))+i-1,t
10120 NEXT i
10130 ON ERROR GOTO 12000
10140 pi = 3.141592
10150 DEF FN rad(d) = d/180*pi
10160 PRINT CHR$(27) "E" CHR$(27) "H"
10170 DATA 197,209,14,115,195,5,0

```

3.5.1.2. Commentary

Line 10030: GSX expects data held in arrays passed to it, to begin at element one in the array, not zero, so the OPTION BASE command is employed to tell Basic to do the same.

Lines 10040-10050: The major arrays declared are as follows:

CODE: Used to define an area of memory which will contain the machine code program to call up GSX.

CTRL: The control array used to tell GSX which function it is to perform.

INT.IN: Used to supply GSX with single values in the range 0-32767.

PTS.IN: Used to supply GSX with pairs of values in the range 0-32767 which define points on the display.

INT.OUT: Used to receive single values from GSX.

PTS.OUT: Used to receive pairs of values defining points on the display from GSX.

Lines 10060-10070: We do not often explicitly declare variables which start off as zero. In this case it is important to set up the variables at the start, since they are going to be altered around the same time that we ask the system where our machine code program is in the memory. Declaring variables for the first time at that point will alter the positioning of the material in memory and result in us trying to run a program which is not there. *Altering* their value will not change the position of items in the memory.

Lines 10080-10120: The values which make up the machine code routine to call GSX are placed into the area of memory defined by the array CTRL. We cannot just place the values straight into CTRL because when they were stored they would each take two bytes, with one of the bytes being zero, and the machine code routine would be a nonsense.

Line 10130: In using GSX it is vital to control any errors which occur during the course of the program, otherwise GSX may not be properly closed and the system can lock up. The error routine at 12000 handles this for us.

Line 10150: GSX works in degrees when specifying angles, while Basic works in radians, the more normal way of measuring angles in technical uses. This function translates from Basic's radians into GSX's degrees.

3.5.2. Call GSX

This module activates the machine code routine to start up GSX and send it whatever information and instructions is contained in the arrays described in the first module.

3.5.2.1. Lines 13000-13050

```
13000 ' *****
13010 ' Call GSX
13020 ' *****
13030 a = VARPTR(code%(1))
13040 CALL a (d%,d%,ctrl%(1),int.in%(1),pts.in%(1,
1),int.out%(1),pts.out%(1,1))
13050 RETURN
```

3.5.3. Close GSX workstation

A standard call to GSX to flush out the contents of any memory buffers it has used and close itself down.

3.5.3.1. *Lines 15000-15060*

```

15000 ' *****
15010 ' Close GSX workstation
15020 ' *****
15030 ctrl%(1) = 2
15040 ctrl%(2) = 0
15050 GOSUB 13000
15060 RETURN

```

3.5.4. Shut down GSX on error

As mentioned in the first module, we must be very careful in handling errors to ensure that whatever else happens, GSX is properly closed down. This module calls up the close routine and then uses a special feature of the ON ERROR command which terminates the program and displays the last Basic error message. The only exception to this is error number four, or 'out of data'. This error is used to control program flow and is handled by other parts of the program without difficulty.

3.5.4.1. *Lines 12000-12060*

```

12000 ' *****
12010 ' Shut down GSX on error
12020 ' *****
12030 IF ERR=4 THEN RESUME NEXT
12040 GOSUB 15000
12050 ON ERROR GOTO 0
12060 STOP

```

3.5.5. Open GSX workstation

A standard call to GSX to set up one of its workstations. In this case, the data in the last line of the module tells GSX to open up the screen as a workstation and to set up the default characteristics of lines, text, fill etc. By altering the device number you can send the output to the printer—see the introduction to this chapter for more details.

3.5.5.1. Lines 14000-14180

```
14000 ' *****
14010 ' Open GSX workstation
14020 ' *****
14030 ctrl%(1) = 1
14040 ctrl%(2) = 0
14050 ctrl%(4) = 10
14060 int.in%(1) = device
14070 RESTORE 14180
14080 FOR i = 2 TO 10
14090   READ int.in%(i)
14100 NEXT i
14110 GOSUB 13000
14120 nchars = int.out%(6)
14130 wide   = int.out%(1)
14140 high   = int.out%(2)
14150 minchar = pts.out%(2,1)
14160 maxchar = pts.out%(2,2)
14170 RETURN
14180 DATA 1,1,1,1,1,1,3,1,1
```

3.5.5.2. Commentary

Lines 14120-14160: After GSX has opened the workstation specified it returns to the program a wide range of information about it. In this case the ones we are interested in are the number of characters which can be printed down the screen, the size of the screen along X and Y axes and minimum and maximum size of the characters that can be displayed.

3.5.6. Update work station

Updating a workstation causes GSX to flush out any buffers associated with the workstation before continuing. This is particularly useful when working with a printer. The close function should flush the buffers, but it is safer to be explicit. In addition, if you wish to output two items in succession it is wise to ensure that all of one has been output before beginning to send the data for the second.

3.5.6.1. Lines 16000-16060

```
16000 ' *****
16010 ' Update GSX workstation
16020 ' *****
```

```

16030 ctrl%(1) = 4
16040 ctrl%(2) = 0
16050 GOSUB 13000
16060 RETURN

```

3.5.7. Set style and index for polygon

A polygon is a closed shape made up from a number of straight lines. On the printer, though not on the screen in the current implementation, GSX is capable of filling such shapes with a variety of different patterns. The style is the overall category such as half-tone, hatch pattern, solid, etc, and the index gives more details of the way the style is to be carried out—if the shape is to be filled with a hatch pattern (ie, lines) are they to be vertical, horizontal, etc.

3.5.7.1. Lines 17000-17110

```

17000 ' *****
17010 ' Set style and index for polygon
17020 ' *****
17030 ctrl%(1) = 23
17040 ctrl%(2) = 0
17050 int.in%(1) = style
17060 GOSUB 13000
17070 ctrl%(1) = 24
17080 ctrl%(2) = 0
17090 int.in%(1) = index
17100 GOSUB 13000
17110 RETURN

```

3.5.8. Draw pie slice

This module represents the first real use of the GSX system in the program. Its purpose is to draw a single slice of the pie chart, as illustrated below:

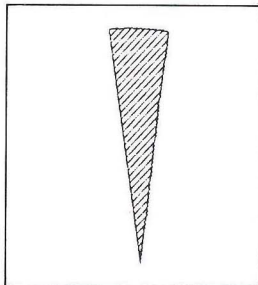


Fig. 3-5: A single pie slice

Anyone who is familiar with GSX will know that a full implementation of the system actually includes an instruction to draw a pie slice. Unfortunately that is not implemented on versions of GSX for the PCW range up to the time of publication. The GSXTEST program at the end of this chapter will tell you whether your particular version of GSX *is* capable of drawing a pie slice without detailed guidance from the program.

3.5.8.1. Lines 18000-18160

```
18000 ' *****
18010 ' Draw pie slice
18020 ' *****
18030 pts.in%(1,1) = x
18040 pts.in%(2,1) = y
18050 count = 2
18060 FOR i = start TO start+angle STEP 5
18070   pts.in%(1,count) = radius*SIN(FN rad(i))+x
18080   pts.in%(2,count) = radius*COS(FN rad(i))+y
18090   count = count+1
18100 NEXT i
18110 pts.in%(1,count) = radius*SIN(FN rad(start+
      angle))+x
18120 pts.in%(2,count) = radius*COS(FN rad(start+
      angle))+y
18130 ctrl%(1) = 9
18140 ctrl%(2) = count
18150 GOSUB 13000
18160 RETURN
```

3.5.8.2. Commentary

Lines 18030-18040: The X and Y co-ordinates represent the centre of the circle into which the pie slice will appear.

Lines 18060-18100: This loop plots series of points five degrees apart around the arc of the pie slice. When displayed on the screen or printer, a series of straight lines between points five degrees apart creates a satisfactory impression of a circle. The points are placed into the PTS.IN array which will be used by GSX in plotting the pie chart. The variable COUNT stores the number of points between which lines will be drawn.

It is worth noting at this point that the calculations here, when eventually applied to every pie slice around the chart, should technically produce a perfect circle and indeed as far as GSX is concerned, they do. Unfortunately,

the way in which the grid of pixels on the PCW's screen is related to GSX's imaginary screen means that the circle on the screen is slightly squashed in height. It is easy enough to correct this by amending lines 18080 and 18120 so that they refer to '1.2*i*radius....' rather than simply 'i*radius'. This will produce a much better circle on the screen, though the labels eventually applied to the chart will point into the segments rather than precisely to the edge.

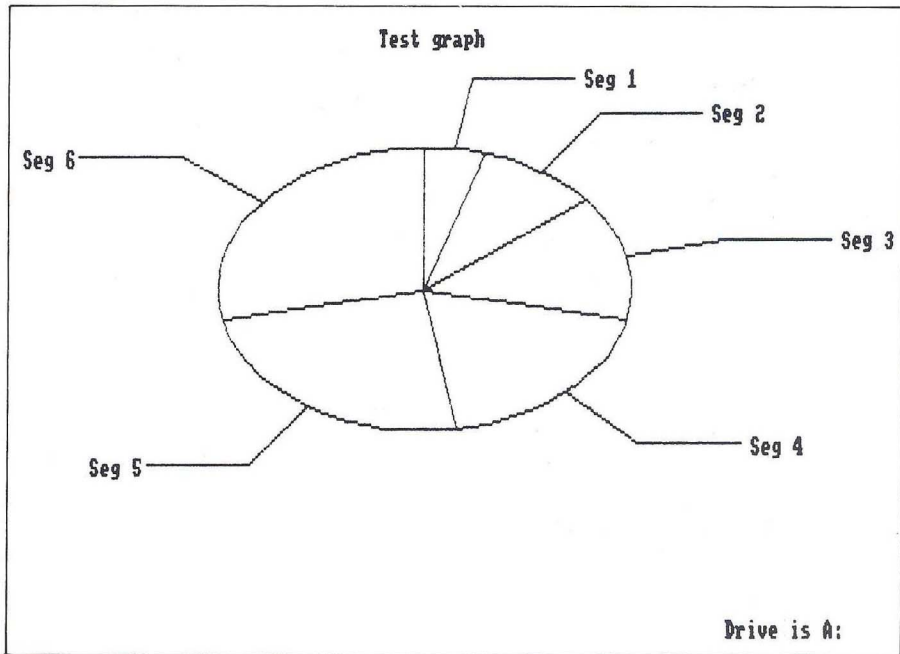


Fig. 3-6: An unadjusted pie-chart on screen

The reason this adjustment is not written into the program is quite simple. When GSX outputs a perfect circle to the printer rather than the screen, the arrangement of dots on the printer leads to the circle being squashed in *width*, the opposite to the screen. This too can be overcome by making a similar alteration to lines 18070 and 18110. Unfortunately you can only bias the circle in one direction—altering both sets of lines would just make the circle bigger.

If one or other means of presentation is much more important to you, you might like to include an adjustment to improve the appearance of the chart but you will have to do it in the knowledge that it will actually appear worse using the other method of output.

Lines 18110-18120: The last point of the arc is plotted outside the loop since it probably not fall at a precise five degree angle.

Lines 18130-18150: The opcode for drawing a polygon is placed into the control array, together with the number of points to be plotted and GSX is called to carry out the drawing.

3.5.9. Set the text height

The height of text to be printed is a straightforward GSX function separate from the actual display of text. This module sets the character size and receives back from GSX information on how many characters the workstation is capable of displaying along the vertical and horizontal axes.

3.5.9.1. Lines 19000-19100

```
19000 ' *****
19010 ' Set the text hight
19020 ' *****
19030 ctrl%(1) = 12
19040 ctrl%(2) = 1
19050 pts.in%(1,1) = 0
19060 pts.in%(2,1) = chigh
19070 GOSUB 13000
19080 cwide = pts.out%(1,1)
19090 chigh = pts.out%(2,1)
19100 RETURN
```

3.5.10. Draw text\$

This module is supplied by the module which calls it with a string called TEXT\$. The characters making up the string are placed into INT.IN and sent to GSX for display, together with an indication of the position the text is to start.

3.5.10.1. Lines 20000-20120

```
20000 ' *****
20010 ' Draw text$
20020 ' *****
20030 ctrl%(1) = 8
20040 ctrl%(2) = 1
20050 ctrl%(4) = LEN(text$)
```

```

20060 FOR i = 1 TO LEN(text$)
20070   int.in%(i) = ASC(MID$(text$,i,1))
20080 NEXT i
20090 pts.in%(1,1) = textx
20100 pts.in%(2,1) = texty
20110 GOSUB 13000
20120 RETURN

```

Lines 20030-20040: The opcode for text printing is placed into the control array. The second figure refers to how many lines must be drawn and must be set to a value of one for text.

Line 20050: In printing text, GSX must be told how many elements of INT.IN will contain valid characters.

Lines 20060-20080: The values of the characters making up the text are placed into INT.IN.

Lines 20090-20100: The X and Y positions at which text is to be placed.

3.5.11. Draw labels

This module places a label against a segment of the chart, using the text height and text drawing modules which you have just entered to do the work. Most of the module is concerned with calculating the correct position for the text.

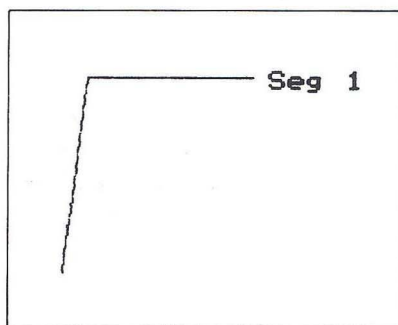


Fig. 3-7: A single label

3.5.11.1. Lines 21000-21310

```
21000 ' *****
21010 ' Draw labels
21020 ' *****
21030 chigh = 400
21040 GOSUB 19000
21050 an = angle/2+start
21060 FOR i = 1 TO 1.5 STEP 0.5
21070   t = i*radius
21080   pts.in%(1,i*2-1) = t*SIN(FN rad(an))+x
21090   pts.in%(2,i*2-1) = t*COS(FN rad(an))+y
21100 NEXT i
21110 IF an<=180 THEN t = radius*0.5 ELSE t = -
      radius*0.5
21120 pts.in%(1,3) = pts.in%(1,2)+t
21130 pts.in%(2,3) = pts.in%(2,2)
21140 ctrl%(1) = 6
21150 ctrl%(2) = 3
21160 GOSUB 13000
21170 WHILE an<=180 AND an>=0
21180   text$ = segname$
21190   textx = 1.5*radius*SIN(FN rad(an))+x+
      radius*0.5+cwide
21200   texty = 1.5*radius*COS(FN rad(an))+y-chigh/
21210   GOSUB 20000
21220   an = -1
21230 WEND
21240 WHILE an>180
21250   text$ = segname$
21260   textx = 1.5*radius*SIN(FN rad(an))+x-
      radius*0.5-cwide*(LEN(text$)+1)
21270   texty = 1.5*radius*COS(FN rad(an))+y-chigh/
21280   GOSUB 20000
21290   an = -1
21300 WEND
21310 RETURN
```

3.5.11.2. Commentary

Lines 21030-21040: The text height is set to 400 GSX units.

Line 21050: The variable ANGLE is supplied by the calling module and represents the angle covered by the segment to be labelled. START represents

the angle at which the segment starts, going clockwise around the chart from a 12 o'clock position.

Lines 21060-21100: This loop calculates two co-ordinates for a line to be drawn between the text of the label and the mid-point of the segment to be labelled. The line will extend out from the circumference of the circle for half a radius.

Line 21110-21120: If you look at the illustration of a chart shown at the beginning of the program, you will see that the line coming straight out from the chart is continued horizontally to the left or right. These lines insert an X co-ordinate which is half a radius to the right or left of the end of the line straight out from the end of the line straight out from the circumference, depending on how far round the circle the label is.

Lines 21140-21160: GSX is called using the polyline command and specifying three points for the two lines to be drawn.

Lines 21170-21300: Printing the text raises the problem of which side of the chart it is to be printed. If it is going to be on the left then it must *end* at the line we have just drawn. If it is on the right hand side then it will *start* at the line—if you are not clear on this, look back on the illustration. These two loops are identical in the way they set the position of the text on the vertical axis and call up GSX to draw the text but one loop sets an X position one character past to the right of the end of the line to the segment, while the other sets the X position to the left of the end by the length of the label plus one character.

3.5.12. Draw pie chart

Having entered all the tools necessary, we can turn to the control module for the program which determines how they are to be used. The responsibilities of this module are to read in the data for the chart, make any calculations about scale, place a title on the screen and then call up the pie slice and labelling modules, supplying them with the information necessary to place their handiwork.

3.5.12.1. Lines 11000-11390

```
11000 ' *****
11010 ' Draw pie chart
11020 ' *****
```



```
11030 RESTORE 22000
11040 READ t$,device,t$,radius,t$,x,t$,y,t$,
    main.title$
11050 GOSUB 14000
11060 RESTORE 22000
11070 READ t$,device,t$,radius,t$,x,t$,y,t$,
    main.title$
11080 total = 0
11090 ERROR(4)
11100 num = 0
11110 WHILE ERL<11120
11120   READ segname$
11130   IF ERL<11120 THEN READ segdata : total =
    total+segdata : num = num+1
11140 WEND
11150 RESTORE 22000
11160 READ t$,device,t$,radius,t$,x,t$,y,t$,
    main.title$
11170 chigh = 800
11180 GOSUB 19000
11190 start = 0
11200 style = 3
11210 index = 4
11220 text$ = main.title$
11230 textx = x-LEN(text$)/2*cwide
11240 texty = y+1.7*radius
11250 GOSUB 20000
11260 FOR seg = 1 TO num
11270   GOSUB 17000
11280   READ segname$,segdata
11290   angle = ROUND(360*segdata/total)
11300   GOSUB 18000
11310   GOSUB 21000
11320   IF index>=6 THEN index = 4 ELSE index =
    index+1
11330   IF (seg=num-1) AND index=4 THEN index = 5
11340   start = start+angle
11350 NEXT seg
11360 IF device<11 THEN WHILE INKEY$="" : WEND
    ELSE GOSUB 16000
11370 GOSUB 15000
11380 LIST 22000-22999
11390 END
```

3.5.12.2. *Commentary*

Lines 11030-11050: The data for the size, position and title for the chart are read into memory and a call is made to GSX to open a workstation using the appropriate device.

Lines 11060-11070: The open workstation module moves the system pointer for the reading of data, so we have to reset it and read past the items we no longer need.

Line 11090: This is a bogus error which is used to set the value of the system variable ERL (errorline) to 11110—the significance of that will become clear in a moment. Note that the error routine at 12000 has been told to ignore error four.

Lines 11100-11140: This loop counts the number of items of data for the chart and produces a total for the data in the variable TOTAL. When the loop runs out of data, the error line variable ERL will be set to 11120 and the loop will terminate.

Lines 11150-11160: Having reset the READ pointer with RESTORE, the first items of data are read again.

Lines 11170-11250: The start angle for the first segment is set, together with the fill characteristics and the main title for the graph is displayed.

Lines 11260-11350: The segments making up the chart are drawn in turn, with the fill characteristics changed for each segment, so that they will be easily distinguishable. The size of each segment is determined by the relationship between the individual item of data for that segment and the total of all the data.

Line 11360-11370: If the chart is being displayed on the screen then the display will continue until a key is pressed. When the display is finished, GSX is instructed to close down the current workstation.

Line 11380: The final act of the program is to display the lines containing the data for the chart.

3.5.13. Data for graph

This is a specimen module containing the data for a test graph. In actual use, this module would be edited and added to in order to reflect the chart you wish to create. The only proviso is that the material should be in the same order as that displayed here.

3.5.13.1. Lines 22000-22130

```
22000 ' *****
22010 ' Data for graph
22020 ' *****
22030 DATA "Device",1
22040 DATA "Radius",7000
22050 DATA "Centre X",16383
22060 DATA "Centre Y",16383
22070 DATA "Main title","Test graph"
22080 DATA "Seg 1",10
22090 DATA "Seg 2",20
22100 DATA "Seg 3",30
22110 DATA "Seg 4",40
22120 DATA "Seg 5",50
22130 DATA "Seg 6",60
```

3.5.13.2. Commentary

Lines 22030: The device on which the chart will be displayed. In normal use this will either be set to one, for the screen or 21/22 for the printer.

Lines 22040-22060: These lines set the centre point and size for the pie chart. The current parameters produce an acceptable size and position, but can be altered if there is good reason.

Line 22070: The main title for the graph, which can be altered to suit the individual chart.

Lines 22080-22130: The data for the individual segments. Note that you do not have to scale the data in any way since the program will allocate to each segment the correct proportion in relation to the sum of all the items of data. The number of items of data you can place here depends on the needs of the individual chart. The only real limitation is that too many segments can create a chart which is impossible to label correctly in the space available. If you do run into problems with labels conflicting with each other, this can often be overcome by rearranging the order of segments.

3.5.13.3. *Testing*

You should now be able to run the program and produce the chart illustrated at the beginning of the program. If so, the program is ready for use.

3.6. Block Graph

The second program in this chapter is a good illustration of the way in which once the basic GSX techniques are mastered, it becomes a simple matter to write new programs using the GSX facilities. The current program produces a three-dimensional graph of the kind illustrated below:

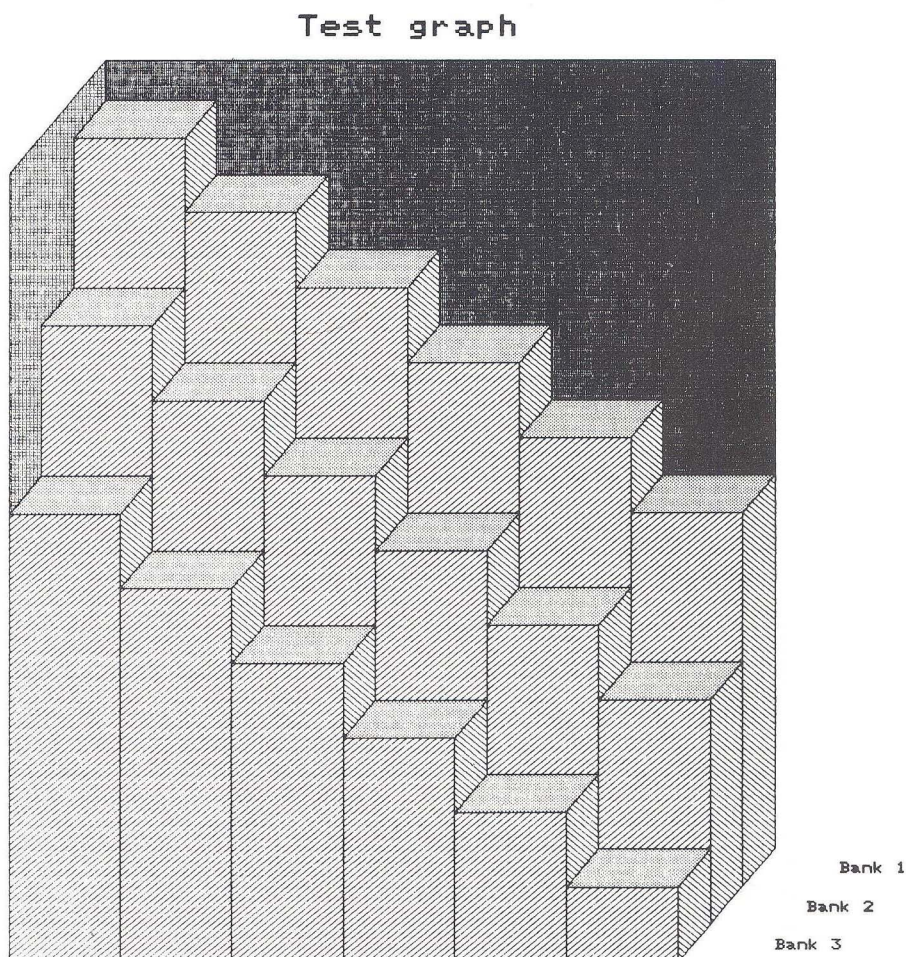


Fig. 3-8: The three-dimensional block graph display on printer

In entering the program you will find that the majority of the material is identical to that found in the last program and, accordingly, we have dispensed with much of the commentary.

3.6.1. Initialise GSX System

This module is practically identical to the module of the same name in the previous program.

3.6.1.1. Lines 10000-10170

```
10000 ' *****
10010 ' Initialise GSX system
10020 ' *****
10030 OPTION BASE 1
10040 DIM code%(10),ctrl%(6),int.in%(80),pts.in%(
    2,75)
10050 DIM int.out%(45),pts.out%(2,75)
10060 d% = 0
10070 a = 0
10080 RESTORE 10170
10090 FOR i = 1 TO 7
10100   READ t
10110   POKE VARPTR(code%(1))+i-1,t
10120 NEXT i
10130 ON ERROR GOTO 12000
10140 pi = 3.141592
10150 DEF FN rad(d) = d/180*pi
10160 PRINT CHR$(27) "E" CHR$(27) "H"
10170 DATA 197,209,14,115,195,5,0
```

3.6.2. GSX control routines

The modules are parallel to those contained in the last program.

3.6.2.1. Lines 12000-17110

```
12000 ' *****
12010 ' Shut down GSX on error
12020 ' *****
12030 GOSUB 15000
12040 ON ERROR GOTO 0
12050 STOP
```

```

13000 ' *****
13010 ' Call GSX
13020 ' *****
13030 a = VARPTR(code%(1))
13040 CALL a (d%,d%,ctrl%(1),int.in%(1),pts.in%(1,
13050 RETURN
14000 ' *****
14010 ' Open GSX workstation
14020 ' *****
14030 ctrl%(1) = 1
14040 ctrl%(2) = 0
14050 ctrl%(4) = 10
14060 int.in%(1) = device
14070 RESTORE 14180
14080 FOR i = 2 TO 10
14090 READ int.in%(i)
14100 NEXT i
14110 GOSUB 13000
14120 nchars = int.out%(6)
14130 wide = int.out%(1)
14140 high = int.out%(2)
14150 minchar = pts.out%(2,1)
14160 maxchar = pts.out%(2,2)
14170 RETURN
14180 DATA 1,1,1,1,1,1,3,1,1
15000 ' *****
15010 ' Close GSX workstation
15020 ' *****
15030 ctrl%(1) = 2
15040 ctrl%(2) = 0
15050 GOSUB 13000
15060 RETURN
16000 ' *****
16010 ' Update GSX workstation
16020 ' *****
16030 ctrl%(1) = 4
16040 ctrl%(2) = 0
16050 GOSUB 13000
16060 RETURN
17000 ' *****
17010 ' Set style and index for polygon
17020 ' *****

```

```
17030 ctrl%(1) = 23
17040 ctrl%(2) = 0
17050 int.in%(1) = style
17060 GOSUB 13000
17070 ctrl%(1) = 24
17080 ctrl%(2) = 0
17090 int.in%(1) = index
17100 GOSUB 13000
17110 RETURN
```

3.6.3. GSX text routines

Once again, these modules are parallel to those contained in the last program.

3.6.3.1. *Lines 23000-24120*

```
23000 ' *****
23010 ' Set the text hight
23020 ' *****
23030 ctrl%(1) = 12
23040 ctrl%(2) = 1
23050 pts.in%(1,1) = 0
23060 pts.in%(2,1) = chigh
23070 GOSUB 13000
23080 cwide = pts.out%(1,1)
23090 chigh = pts.out%(2,1)
23100 RETURN
24000 ' *****
24010 ' Draw text$
24020 ' *****
24030 ctrl%(1) = 8
24040 ctrl%(2) = 1
24050 ctrl%(4) = LEN(text$)
24060 FOR i = 1 TO LEN(text$)
24070 int.in%(i) = ASC(MID$(text$,i,1))
24080 NEXT i
24090 pts.in%(1,1) = textx
24100 pts.in%(2,1) = texty
24110 GOSUB 13000
24120 RETURN
```

3.6.4. Set PTS.IN for front/top/side

There is nothing complex about these three modules, all that they do is to set up a series of four points in the PTS.IN array which allow a four sided polygon representing either the front, top or side of one of the blocks which make up the graph to be plotted.

3.6.4.1. Lines 19000-21080

```

19000 ' *****
19010 ' Set ptsin for front
19020 ' *****
19030 pts.in%(1,1) = x : pts.in%(2,1) = y
19040 pts.in%(1,2) = x : pts.in%(2,2) = y+high
19050 pts.in%(1,3) = x+wide : pts.in%(2,3) = y+
    high
19060 pts.in%(1,4) = x+wide : pts.in%(2,4) = y
19070 index = 4 : style = 3
19080 RETURN
20000 ' *****
20010 ' Set ptsin for top
20020 ' *****
20030 pts.in%(1,1) = x : pts.in%(2,1) = y+high
20040 pts.in%(1,2) = x+deepx : pts.in%(2,2) = y+
    high+deepy
20050 pts.in%(1,3) = x+wide+deepx : pts.in%(2,3) =
    y+high+deepy
20060 pts.in%(1,4) = x+wide : pts.in%(2,4) = y+
    high
20070 index = 3 : style = 2
20080 RETURN
21000 ' *****
21010 ' Set ptsin for side
21020 ' *****
21030 pts.in%(1,1) = x+wide : pts.in%(2,1) = y
21040 pts.in%(1,2) = x+wide : pts.in%(2,2) = y+
    high
21050 pts.in%(1,3) = x+wide+deepx : pts.in%(2,3) =
    y+high+deepy
21060 pts.in%(1,4) = x+wide+deepx : pts.in%(2,4) =
    y+deepy
21070 index = 3 : style = 3
21080 RETURN

```


3.6.4.2. *Commentary*

Line 19030: The variables X and Y are used throughout these three modules to refer to the horizontal and vertical position of the start point of the block.

Line 19040: The variable HIGH represents the basic height of the block, as determined by the item of data being presented.

Line 19050: The variable WIDE is used to refer to the width of the block, as determined by the number of blocks to be displayed compared to the width of the whole chart.

Line 19070: Each module ends with a line like this which sets the fill characteristics for the polygon making up the particular face. The fill characteristics are only relevant when the graph is being displayed on the printer, since the current implementation on the PCW range does not provide for polygon filling on the screen. The subsequent GSX test program will allow you to interrogate GSX to find out if the version with which you have been supplied is capable of creating fill patterns on the screen.

Line 20040: DEEPX and DEEPY refer to the added distance along the X and Y axes used to define the position of the back of the block.

3.6.5. Draw a polygon

Once the points which define one face of the block have been defined by one or other of the three previous modules, this module first draws a polygon with the relevant fill characteristic using the GSX 'draw polygon' command.

As in the last program, providing that the fill characteristic is set correctly, drawing a polygon will automatically result in a filled area, though this will only be apparent on the printer using the current implementation of GSX on the PCW range.

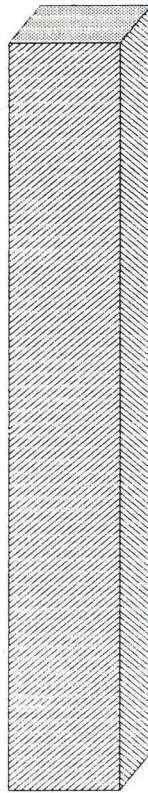


Fig. 3-9: A single block made up of polygons

3.6.5.1. Lines 22000-22070

```

22000 ' *****
22010 ' Draw a polygon
22020 ' *****
22030 GOSUB 17000
22040 ctrl%(1) = 9
22050 ctrl%(2) = 4
22060 GOSUB 13000
22070 RETURN

```

3.6.6. Draw single block

A simple module that uses the preceeding modules to draw a single block of the graph.

3.6.6.1. Lines 18000-18060

```
18000 ' *****
18010 ' Draw block
18020 ' *****
18030 GOSUB 19000 : GOSUB 22000
18040 GOSUB 20000 : GOSUB 22000
18050 GOSUB 21000 : GOSUB 22000
18060 RETURN
```

3.6.7. Draw bar graph

As with the last program, this is the module which controls the flow of the program and performs the calculations on the scaling of data and the positioning of individual blocks within the overall graph.

3.6.7.1. Lines 11000-11580

```
11000 ' *****
11010 ' Draw bar graph
11020 ' *****
11030 RESTORE 25000
11040 READ t$,device,t$,startx,t$,starty,t$,maxx,
    t$,maxy,t$,main.title$,t$,nbanks,t$,nbars
11050 big = 0
11060 FOR bank = 1 TO nbanks
11070   READ t$
11080   FOR bar = 1 TO nbars
11090     READ t
11100     big = MAX(big,t)
11110   NEXT bar
11120 NEXT bank
11130 GOSUB 14000
11140 RESTORE 25000
11150 READ t$,device,t$,startx,t$,starty,t$,maxx,
    t$,maxy,t$,main.title$,t$,nbanks,t$,nbars
11160 chigh = 800
11170 GOSUB 23000
11180 d = 5000
11190 an = 0.7854
11200 deepx = d*COS(an) : deepy = d*SIN(an)
```

```
11210 x = startx : y = starty
11220 wide = 0 : high = maxy-deepy
11230 GOSUB 21000 : index = 4 : style = 2
11240 GOSUB 22000
11250 wide = maxx-deepx
11260 x = startx+deepx : y = starty+deepy
11270 GOSUB 19000 : index = 5 : style = 2
11280 GOSUB 22000
11290 scale = INT(0.95*high/big)
11300 text$ = main.title$
11310 textx = startx+(maxx-LEN(text$)*cwide)/2
11320 texty = starty+maxy+chigh
11330 GOSUB 24000
11340 y = starty+INT(d*COS(an))
11350 FOR bank = 1 TO nbanks
11360   IF bank<>nbanks THEN deep = INT(d/nbanks)
      ELSE deep = d-INT(d/nbanks)*(nbanks-1)
11370   deepx = INT(deep*SIN(an))
11380   deepy = INT(deep*COS(an))
11390   y = y-deepy
11400   IF bank=nbanks THEN x = startx ELSE x =
      startx+(nbanks-bank)*deepx
11410   chigh = 400
11420   GOSUB 23000
11430   textx = x+maxx
11440   texty = y+deepy/2-chigh/2
11450   READ text$
11460   GOSUB 24000
11470   FOR bar = 1 TO nbars
11480     READ bank.high
11490     high = bank.high*scale
11500     IF bar<>nbars THEN wide = INT((maxx-d*COS(
      an))/nbars) ELSE wide = maxx-d*COS(an)-INT((
      maxx-d*COS(an))/nbars)*(nbars-1)
11510     GOSUB 18000
11520     x = x+wide
11530   NEXT bar
11540 NEXT bank
11550 IF device<11 THEN WHILE INKEY$="" : WEND
      ELSE GOSUB 16000
11560 GOSUB 15000
11570 LIST 25000-25999
11580 END
```


3.6.7.2. *Commentary*

Lines 11030-11040: The data defining the position, overall size and title to applied to the graph are read in from the DATA lines at the end of the program. The main variables are:

DEVICE: The number of the device on which the graph is to be displayed.

STARTX and STARTY: The position of the origin of the graph.

MAXX and MAXY: The length of X and Y axes.

MAIN.TITLE\$: The title to be applied to the graph.

NBANKS: The number of rows or 'banks' to be displayed on the graph.

NBARS: The number of blocks along each bank.

Lines 11050-11120: The data for the individual blocks is read in from the DATA lines and the largest item, on the basis of which everything else will be scaled, is placed into the variable BIG.

Line 11150: Having reset the data pointer, the main items are read in again so that the pointer is correctly positioned to pick up the data for the individual blocks.

Lines 11160-11170: The character height is set for the title.

Lines 11180-11240: The graph, as can be seen from the illustration at the beginning of this section, is displayed with a background of two walls, one on the left of the display and one behind it. These lines calculate the points for the left hand wall of the framework and then call upon the relevant drawing modules to create it. The variable AN represents the angle of slope of the tops of the blocks and D holds the overall depth of the graph in GSX units.

Lines 11250-11280: The back wall of the framework is calculated and drawn.

Line 11290: The scaling factor for the graph is calculated so that the largest item in the graph will not exceed 95% of the height of the surrounding walls.

Lines 11300-11330: The main title of the graph is displayed.

Line 11340: The position vertically of the rear bank of the graph.

Lines 11350-11540: The overall loop which draws the banks, starting with the rearmost bank, so that visual priority is given to the front-most.

Line 11360: The variable DEEP, which represents the distance which the individual block will appear to go 'backwards' into the framework. On the final bank, the one at the front, any adjustment necessary to take account of the fact that DEEP has to be an integer number for the other banks is made.

Lines 11370-11380: These two variables contain the offsets which define the position of the front of the block, compared to the back.

Line 11390: The vertical position of the front of the block consists of the position of the back, minus the value of DEEPY.

Line 11400: Once again, since we shall drawing banks which are offset horizontally each time by an integer number, it is possible that there will be a rounding error in the positioning of the banks. This is only important in the case of the front bank, which has to be aligned with the left-hand side of the framework, so an adjustment is made for the last bank to be drawn.

Lines 11410-11460: The label for the particular bank is displayed at the right-hand side of the framework.

Lines 11470-11530: This loop reads in the data for the height of individual blocks within a bank and calls up the drawing modules to place them on the screen. The raw data is scaled according to the maximum item which is going to have to be displayed. The width of each block is normally WIDE units but since this is an integer number which may not divide exactly into the total width of the graph, the width of the last block is adjusted so that it precisely matches the edge of the framework.

3.6.8. Data for graph

As with the last program, the data on which the graph is based is held in DATA statements at the end of the program. The data can be edited to produce a display of any chosen data.

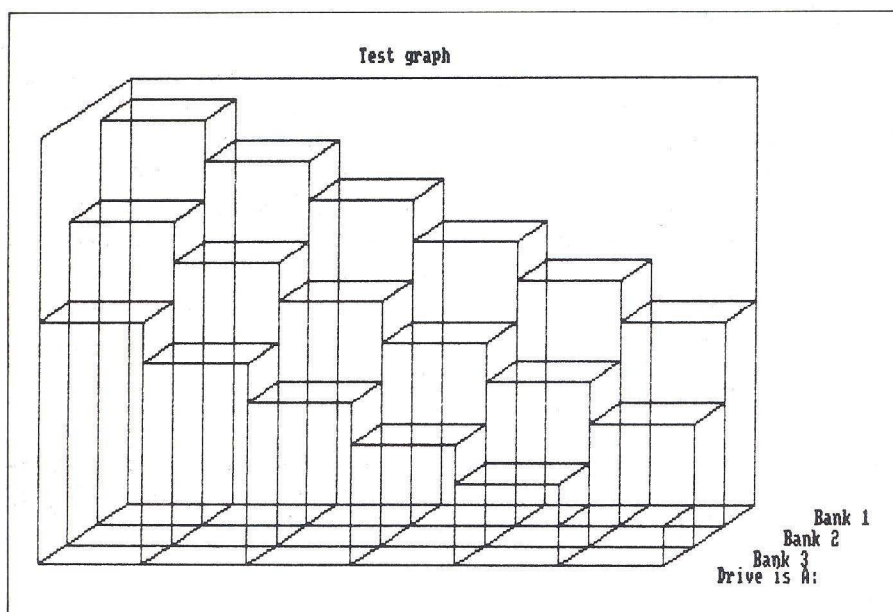


Fig. 3-10: The test graph as seen on screen

3.6.8.1. Lines 25000-25130

```

25000 ' *****
25010 ' Data for graph
25020 ' *****
25030 DATA "Device",1
25040 DATA "Start X",0
25050 DATA "Start Y",0
25060 DATA "Maximum X",28000
25070 DATA "Maximum Y",28000
25080 DATA "Main title","Test graph"
25090 DATA "Number of banks",3
25100 DATA "Number of elements per bank",6
25110 DATA "Bank 1",100,90,80,70,60,50
25120 DATA "Bank 2",80,70,60,50,40,30
25130 DATA "Bank 3",60,50,40,30,20,10

```

3.6.8.2. Commentary

Line 26030: The number of the device on which output is to be made.

Lines 26040-26070: These lines define a reasonable start point and size for a typical graph, but can be altered if there is good reason.

Line 26080: The title of the graph.

Lines 26090-26100: Since the items of data may fall into groups representing different banks, the program needs to be told how many banks and how many items there are in each bank before it can decide how to read the data. The line representing each bank begins with the label to be attached to the bank, followed by the data for each block within it. It is important that having entered the figure for the number of banks and items within them that these lines conform to what the program has been told. If this is not done, the graph will either produce a nonsense and/or the program will stop with an error as it tries to read data which does not exist.

3.6.8.3. Testing

You should now be able to run the program and produce the illustration shown at the beginning of this section. If so, the program is ready to display your own charts once you have edited the data module at the end.

3.7 GSX Test Program

The purpose of this program is to allow you to use the facilities built into GSX that inform the user of the capabilities of the system. If, in the future, the version of GSX supplied with the system is upgraded or upgraded versions become available as separate items, this program can be used to display the capabilities of the new version without being limited to documentation which may not always be up to date with the latest developments. The program works on the basis of individual devices, since the capabilities of the system will be different depending on the device being employed for output. On the standard system, for instance, far more capabilities are available when using the printer than the screen.

Though the program is not short, it is nevertheless simple, with the biggest part of the program devoted to displaying the information obtained on the screen. Much of the information which the program provides is fairly technical, but those who wish to go further with GSX will nevertheless find it invaluable. As we pointed out in the introduction to this chapter, full details of how to make use of the GSX system are included in the GSX Manual available from Digital Research, the designers of GSX.

The program also gives a useful example of the way information is received back from GSX in the arrays INT.OUT and PTS.OUT.

3.7.1. Initialise GSX system

This module is parallel to the initialisation modules of the previous two programs.

3.7.1.1. Lines 10000-10150

```
10000 ' *****
10010 ' Initialise GSX system
10020 ' *****
10030 OPTION BASE 1
10040 DIM code%(10),ctrl%(6),int.in%(80),pts.in%(
2,75)
10050 DIM int.out%(45),pts.out%(2,75)
10060 d% = 0
10070 a = 0
10080 RESTORE 10150
10090 FOR i = 1 TO 7
10100 READ t
10110 POKE VARPTR(code%(1))+i-1,t
10120 NEXT i
10130 ON ERROR GOTO 12000
10140 cls$ = CHR$(27)+"E"+CHR$(27)+"H"
10150 DATA 197,209,14,115,195,5,0
```

3.7.2. Various GSX control routines and error handling

A series of modules which parallel items in previous programs. In this particular program we shall not actually be making use of the graphics facilities of GSX so we need only a limited range of modules.

3.7.2.1. Lines 12000-15060

```
12000 ' *****
12010 ' Shut down GSX on error
12020 ' *****
12030 GOSUB 15000
12040 ON ERROR GOTO 0
12050 STOP
13000 ' *****
13010 ' Call GSX
13020 ' *****
13030 a = VARPTR(code%(1))
```

```

13040 CALL a (d%,d%,ctrl%(1),int.in%(1),pts.in%(1,
1),int.out%(1),pts.out%(1,1))
13050 RETURN
14000 ' *****
14010 ' Open GSX workstation
14020 ' *****
14030 ctrl%(1) = 1
14040 ctrl%(2) = 0
14050 ctrl%(4) = 10
14060 int.in%(1) = device
14070 RESTORE 14180
14080 FOR i = 2 TO 10
14090 READ int.in%(i)
14100 NEXT i
14110 GOSUB 13000
14120 pie = 0
14130 FOR i = 16 TO 25
14140 IF int.out%(i)=3 THEN pie = -1
14150 IF int.out%(i)<0 THEN i = 26
14160 NEXT i
14170 RETURN
14180 DATA 1,1,1,1,1,1,3,1,1
15000 ' *****
15010 ' Close GSX workstation
15020 ' *****
15030 ctrl%(1) = 2
15040 ctrl%(2) = 0
15050 GOSUB 13000
15060 RETURN

```

3.7.3. Print data item and Data

These two modules, which are used by the control module which follows performs the task of translating a number into a name taken from a list stored in the form of DATA statements and then printing out that name at the current cursor position.

3.7.3.1. Lines 16000-17050

```

16000 ' *****
16010 ' Print data item
16020 ' *****
16030 FOR i = 1 TO item

```

```
16040 READ item$
16050 NEXT i
16060 PRINT #1,item$
16070 WHILE item$<>"end"
16080 READ item$
16090 WEND
16100 RETURN
17000 ' *****
17010 ' Data
17020 ' *****
17030 DATA "OUTPUT","INPUT","OUTPUT AND INPUT",
        DEVICE INDEPENDENT SEGMENT STORAGE","GKS
        METAFILE OUTPUT","end"
17040 DATA "MONOCHROME","COLOR","end"
17050 DATA "BLACK AND WHITE","BLACK,RED,GREEN,
        BLUE,CYAN,YELLOW,MAGENTA,WHITE","end"
```

3.7.4. Print out GSX

The control module for the program which accepts from the user the device for which data is to be printed out and interrogates the GSX system to obtain the data.

3.7.4.1. Lines 11000-11870

```
11000 ' *****
11010 ' Print out GSX
11020 ' *****
11030 found = 0
11040 WHILE NOT found
11050 PRINT cls$
11060 INPUT "Number of device to test ",device
11070 OPEN "i",1,"assign.sys"
11080 WHILE NOT EOF(1) AND NOT found
11090 LINE INPUT #1,t$
11100 WHILE LEFT$(t$,1) = " "
11110 t$=MID$(t$,2)
11120 WEND
11130 num = 0
11140 n$ = "0"
11150 WHILE n$>="0" AND n$<="9"
11160 num = num*10+ASC(n$)-48
11170 n$ = LEFT$(t$,1)
```

```
11180     t$ = MID$(t$,2)
11190     WEND
11200     found = num=device
11210     WEND
11220     CLOSE 1
11230     IF NOT found THEN PRINT CHR$(7) "Device"
        device " is not defined in ASSIGN.SYS -
        press a key" : WHILE INKEY$="" : WEND
11240 WEND
11250 INPUT "Send output to screen or printer (S/
P)";p$
11260 p$ = LOWER$(p$)
11270 GOSUB 14000
11280 RESTORE 17000
11290 OPEN "o",1,"temp$$$$.$$$"
11300 PRINT #1,"Device driver " t$ : PRINT #1
11310 PRINT #1,"1) Device information"
11320 PRINT #1,SPC(10) "a) Printing area" int.out%
        (1) "x" int.out%(2) "(width x height) in
        pixels"
11330 PRINT #1,SPC(10) "b) Device type "; : item =
        int.out%(45)+1 : GOSUB 16000
11340 PRINT #1,
11350 PRINT #1,"2) Color"
11360 PRINT #1,SPC(10) "a) This device displays
        in "; : item = int.out%(36)+1 : GOSUB 16000
11370 PRINT #1,SPC(10) "b) Colors available are: "
        ; : item = int.out%(36)+1 : GOSUB 16000
11380 PRINT #1,
11390 PRINT #1,"3) Characters"
11400 PRINT #1,SPC(10) "a) Number of character
        fonts is:" int.out%(11)
11410 PRINT #1,SPC(10) "b) Number of character
        sizes:";
11420 IF int.out%(6)=0 THEN PRINT #1," Continuous
        scaling" ELSE PRINT #1,int.out%(6)
11430 PRINT #1,SPC(10) "c) Text rotation can ";
11440 IF int.out%(37)=0 THEN PRINT #1,"not ";
11450 PRINT #1,"be used"
11460 PRINT #1,SPC(10) "d) Minimum character
        height:" pts.out%(2,1) "(NDC units)"
11470 IF int.out%(6)<>1 THEN PRINT #1,SPC(10) "e)
        Maximum character height:" pts.out%(2,2) "(
```



```
      NDC units)"
11480 PRINT #1
11490 PRINT #1,"4) Lines"
11500 PRINT #1,SPC(10) "a) Number of line types:"
      int.out%(7)
11510 PRINT #1,SPC(10) "b) Number of line widths:"
      int.out%(8)
11520 PRINT #1,SPC(10) "c) Minimum line width:"
      pts.out%(1,3) "(NDC units)"
11530 IF int.out%(8)>1 THEN PRINT #1,SPC(10) "d)
      Maximum line width:" pts.out%(1,4) "(NDC
      units)"
11540 PRINT #1
11550 PRINT #1,"5) Markers"
11560 PRINT #1,SPC(10) "a) Number of marker types:
      " int.out%(9)
11570 PRINT #1,SPC(10) "b) Number of marker sizes:
      " int.out%(10)
11580 PRINT #1,SPC(10) "c) Minimum marker size:"
      pts.out%(2,5) "(NDC units)"
11590 IF int.out%(10)>1 THEN PRINT #1,SPC(10) "d)
      Maximum marker size:" pts.out%(2,6) "(NDC
      units)"
11600 PRINT #1
11610 PRINT #1,"6) General Drawing Primitive"
11620 PRINT #1,SPC(10) "a) GDP's available: ";
11630 FOR i = 16 TO 25
11640   IF int.out%(i)=1 THEN PRINT #1,"BAR ";
11650   IF int.out%(i)=2 THEN PRINT #1,"ARC ";
11660   IF int.out%(i)=3 THEN PRINT #1,"PIE SLICE ";
11670   IF int.out%(i)=4 THEN PRINT #1,"CIRCLE ";
11680 NEXT i
11690 PRINT #1 : PRINT #1
11700 PRINT #1,"7) Fill patterns"
11710 PRINT #1,SPC(10) "a) Number of patterns: "
      int.out%(12)
11720 PRINT #1,SPC(10) "b) Number of hatches: "
      int.out%(13)
11730 PRINT #1,SPC(10) "c) Area fill ";
11740 IF int.out%(38)=0 THEN PRINT #1,"not ";
11750 PRINT #1,"available"
11760 PRINT #1
11770 GOSUB 15000
```

```

11780 CLOSE 1
11790 PRINT c1s$
11800 OPEN "i",1,"temp$$$$.$$$"
11810 WHILE NOT EOF(1)
11820   LINE INPUT #1,t$
11830   IF p$="p" THEN LPRINT t$ ELSE PRINT t$
11840 WEND
11850 CLOSE 1
11860 KILL "temp$$$$.$$$"
11870 END

```

3.7.4.2. *Commentary*

Lines 11040-11240: This loop accepts from the user the number of the device about which GSX is to be interrogated. The devices of which the system is aware at the present moment are already listed, by number, in the file ASSIGN.SYS. What these lines do is to scan through the ASSIGN.SYS file to check that the device number specified by the user is registered in the file. If it is not, an error message will be displayed.

Line 11270: The module at 14000 is called to open a workstation on the device which the user has specified. In this case, the effect we are interested in is that GSX immediately fills the arrays INT.OUT and PTS.OUT with a collection of values which represent messages from GSX describing what it can do with the specified device.

Line 11290: Since the output of the program can be sent either to the screen or the printer, it is easier to create a disk file of the output first and send it to the appropriate device later.

Line 11320: Each device has a maximum area on which printing can take place, expressed in pixels, or individual dots. This line displays the values for the two dimensions.

Line 11330: The device which is being considered can be one of several types. It can be:

- a) An output device like the screen or printer.
- b) An input device like the keyboard.
- c) An input/output device like a disk drive.
- d) 'Device independent segment storage', which is not implemented on the PCW system.
- e) GKS Metafile output—a special file on disk containing GSX information which can be used to create a display of some kind.

Note that these types do not necessarily tell you exactly what the device is, only the broad category.

Lines 11360-11370: Depending on the device, GSX is capable of supplying its output either in monochrome or colour. This will not be of much interest in relation to the monochrome display of a standard PCW system but other devices, such as colour printer or plotters, may make use of the facility.

Lines 11390-11470: A series of items relating to the text capabilities when using the device.

Lines 11490-11530: GSX is in principle capable of drawing a variety of different types of lines, such as dotted lines, solid and so on. This section of the program displays how many are available on a particular device.

Lines 11550-11590: The variety of different markers which can be placed on the screen. The markers available at the time of publication are listed under the Set Polymarker command in the table at 3.4.

Lines 11610-11680: Apart from the ability to define a shape by supplying GSX with the positions of its vertices or corners, the full GSX system is also capable of automatically drawing a number of shapes on the basis of very limited information. These shapes are called 'primitives'. The GSX supplied as standard with the PCW range includes only the primitive for a bar; these lines will tell you whether your version can do any more with the particular device.

Lines 11700-11750: These lines give information on the fill patterns which are available for the device.

Lines 11800-11840: The file into which the information has been placed is read back from the disk and output to either to the screen or printer, according to the earlier choice made by the user.

3.7.4.3. *Testing*

Running the program should now result in the creation of a display something like that shown overleaf, which was created using this program. The capabilities of the program in relation to devices other than the screen or printer can, of course, only be tested if you have added other devices to your system. Even if you only have the basic system, it is worth checking your version of GSX to see if any improvements have been made.

3.7.5. A specimen GSX__TEST output

The following table represents the output of the GSX__TEST program in relation to the screen driver:

Device driver @:DDSCREEN

1) Device information

- a) Printing area 719 x 247 (width x height) in pixels
- b) Device type OUTPUT

2) Color

- a) This device displays in MONOCHROME
- b) Colors available are: BLACK AND WHITE

3) Characters

- a) Number of character fonts is: 1
- b) Number of character sizes: 1
- c) Text rotation can not be used
- d) Minimum character height: 925 (NDC units)

4) Lines

- a) Number of line types: 5
- b) Number of line widths: 1
- c) Minimum line width: 46 (NDC units)

5) Markers

- a) Number of marker types: 5
- b) Number of marker sizes: 1
- c) Minimum marker size: 925 (NDC units)

6) General Drawing Primitive

- a) GDP's available: BAR

7) Fill patterns

- a) Number of patterns: 0
- b) Number of hatches: 0
- c) Area fill not available

CHAPTER 4

Logo

Designed in the 1970s, the germ of the Logo idea came from the visionary educationalist Dr. Seymour Papert. It was intended to be, and according to many of its advocates it *is* the ultimate language for learning computing—or even learning to think.

There are many reasons for the sometimes exaggerated claims made for Logo, but perhaps its greatest claim to fame is the ‘turtle’, the small arrow-shaped cursor which can be used to draw complex shapes on the basis of simple English commands. Papert called the turtle ‘an object to think with’, and there is no doubt that the ease with which drawing processes, at least, can be visualised in Logo, has opened up new possibilities to programmers all over the world.

In addition to its graphical abilities, Logo is the most easily available of the ‘list processing’ languages. List processing is another programming cult, much beloved of those involved in artificial intelligence experiments, but it is also a useful and flexible language feature that anyone can make use of. In essence, a language capable of list processing is capable of accepting lists of items as inputs and processing them, without first being told exactly what the structure of the list is going to be. In some circumstances a Logo program which asks for an input can be given a string, a number, or a list of strings or a list of numbers and it should not reject any of them because they are the wrong ‘type’ of data. In a world where reality often stubbornly refuses to conform to the neat categories that some other languages need to impose on data before they will work, Logo’s list processing can come as a breath of fresh air.

In one short chapter it is clearly not going to be possible to even begin to exhaust the capabilities of the PCW’s Logo. What we can show, however, especially for those who have perhaps made little or no use of Logo in the past, is just how easily simple applications can be written in Logo and at the same time how useful a simple Logo application can be. The rest of this chapter is given over to a listing and brief description of two Logo programs which produce attractively formatted graphs and a third which makes effective use of list processing. We shall not try to duplicate the detailed descriptions of Logo commands contained in the manual which accompanies the PCW—our aim is

solely to whet your appetite and to convince you that Logo on the PCW can be both fun and useful.

4.1. Running Logo

In order to run the Logo language you will need to ensure that you have a disc containing the following files:

LOGO.COM
SUBMIT.COM
LOGO.SUB
KEYS.DRL
SETKEYS.COM

All of these files are on the Logo and CP/M master discs supplied with your system. The language can be run either from a working disc or from a single purpose disc, both of which are described in the chapter on CP/M. The language itself is run by entering the command 'LOGO' on the CP/M command line.

It is important to remember that the Logo submit file installs a new keyboard designed especially for the language but does not remove the new keyboard when you quit Logo. As a result of this, the cursor control and DEL—> keys do not function correctly after running Logo. The solution to this is to either reset the machine or to use the instructions given in the manual in the section on SETKEYS to create a special file containing the normal keyboard and to add at the end of the submit file a command to re-install the keyboard it represents.

4.2. Entering the programs

The way in which Logo programs are entered into your machine depends to some extent upon the nature of the language. In Logo, all of the work of a program is done by means of 'procedures'. Some of these procedures are built into the language and are called 'primitives'. Primitives are the keywords you will find described in your Logo manual. When you write a program you use these procedures to write 'bigger' procedures, but in many ways Logo makes no distinction between what it has built in and what you have created for yourself.

A procedure is simply a piece of software which tells Logo how to do something and which has a name. When you load a program into the system, you will see a series of messages telling you that procedures have been defined, ie, that Logo knows what to do if the particular procedure name is used as a keyword in a program. A Logo program is a collection of procedures held in memory in no particular order.

When you write Logo programs, it is often advantageous to make use of the same procedures in more than one program in the same way that you make use of primitives in more than one program—the first two programs in the chapter show this in action. Clearly it would be a waste of time to enter identical procedures twice over, so Logo provides you with a way to save selected procedures from one program for later use with another. The way in which this is done is to use the ER command to delete from the program all the procedures which you do not wish to save and then to save the remaining procedures under the name of the new program you wish to develop.

The programs given in the remainder of this chapter are listed in the form of procedures, each numbered individually rather than in an overall sequence within the program. You can enter the procedures in any order provided that you do not miss any out. The numbers in the listings are included purely as a reference aid to make it easier to read the program alongside the commentary, they should *not* be entered along with the line since Logo will make no sense of them.

Finally, as with Basic, it is wise to save the program you are entering regularly, preferably as each procedure is completed. Logo will refuse to overwrite a previous version of a program so the only way to carry out regular saving is to rename the program each time you save it, eg, GRAPH.001, GRAPH.002, etc. Deleting the existing copy of a program before re-saving it under the same name is *not* recommended since it leaves you, for however short a time, with no copy of the program securely on the disc.

4.3. L__Graph

Our first Logo program produces a simple line graph which illustrates some of the potential and indeed, some of the limitations of the implementation of the language on the PCW range. The program shows the way in which a Logo program needs to be structured, even more than Basic, into functional modules called procedures. It also illustrates the fact that facilities that would be thought of as standard in other languages have to be specially written.

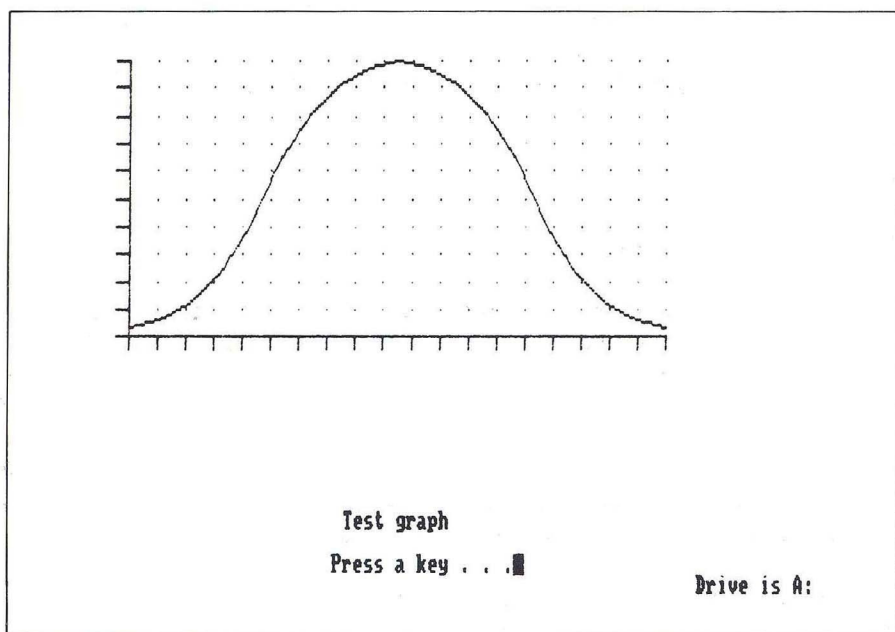


Fig. 4-1: Typical output of L_GRAPH

4.3.1. Linegraph

This is the overall control module for the program. It accepts a mass of data from the calling procedure and then, in its turn, calls a series of procedures which draw a regular grid and axes, place a graph on the grid and label it.

4.3.1.1. Listing

```
1:   to linegraph :title :size :data
2:   (local "pos "loop "t)
3:   setsplit 3
4:   ss
5:   ct
6:   make "pos (list first tf item 2 tf)
7:   ht
8:   pu
9:   htext :title 28
10:  grid first :size item 2 :size count :data
    item 3 :size
11:  dograph :size :data
```

```
12:  htext [Press a key . . .] 30
13:  label "loop
14:  if keyp [make "t rc] [go "loop]
15:  setsplit 10
16:  ct
17:  end
```

4.3.1.2. *Commentary*

Line 1: Logo procedures are not simply keywords in the same way as Basic keywords. All Logo procedures are, at least potentially, *functions*. What that means is that a procedure, when it is called up by its name, can also be sent a number of values or strings, can manipulate the data it has been sent and, if necessary, return the result of that manipulation to whichever part of the program called the procedure. The variables that the procedure can accept when it is called are always made clear in the first line of the procedure, where the names under which the data is accepted must be listed.

The variables supplied to the `L__GRAPH` by the procedure which starts it up are:

TITLE: A heading to be printed on the screen above the graph.

SIZE: The length, in turtle units, of the X axis, the Y axis and the length of a single unit on the Y axis.

DATA: A list of numeric data representing successive values for the graph—the units are irrelevant since the program will perform its own scaling.

Lines 3-5: The program works with three lines of text screen at the bottom, rather than the default of 10.

Line 6: The list `POS` is used to store the X and Y co-ordinates of the turtle before any other procedures are called up.

Line 9: A call to a later procedure which places text onto the screen horizontally.

Line 10: A call to the procedure which draws the grid onto which the graph will be placed

Line 11: A call to the procedure which draws the graph itself.

Lines 12-14: A call to the HTEXT procedure to print a prompt at the bottom of the screen, followed by a waiting state which is terminated when a key is pressed.

4.3.2. Grid

This procedure draws the axes of the graph and creates a regular grid on the screen which makes the graph itself easier to read.

4.3.2.1. Listing

```
1:   to grid :xsize :ysize :xcount :ycount
2:   seth 0
3:   pu
4:   setpos :pos
5:   fd :ysize
6:   rt 90
7:   bk 10
8:   repeat :ycount [pd do_grid :xsize + 10 :
   ysize / :ycount]
9:   pu
10:  setpos :pos
11:  seth 0
12:  repeat :xcount [pd bk 10 fd 10 pe do_grid :
   ysize :xsize / (:xcount - 1)]
13:  setpos :pos
14:  pu
15:  fd :ysize
16:  rt 90
17:  repeat :ycount [px do_grid :xsize :ysize / :
   ycount]
18:  setpos :pos
19:  pd
20:  bk 10
21:  fd :xsize + 10
22:  bk :xsize
23:  lt 90
24:  fd :ysize
25:  end
```

4.3.2.2. *Commentary*

Line 1: The variables passed to the procedure are:

XSIZE: The length in turtle units of the X (horizontal) axis.

YSIZE: The length in turtle units of the Y (vertical) axis.

XCOUNT: The number of divisions to be marked on the X axis.

YCOUNT: The number of divisions to be marked on the Y axis.

Lines 2-7: The turtle is moved to the top of the Y axis and then 10 turtle units to the left, all with the pen up.

Line 8: The procedure DO_GRID, which you have not yet entered, is carried out as many times as there are divisions on the Y axis, each time with the pen down. The effect of DO_GRID is to draw a line across the screen, then to shift the turtle back to the start of the line and move down. When called from this repeat loop the turtle is pointing to the right of the screen, so the lines are drawn horizontally.

When you come to enter DO_GRID it is worth flicking back to this point to compare the name of the variables which are sent to the module and the names which are received—you will see that they are not the same. It is always important to remember in Logo that when you send variables to a procedure it is not the variables themselves that are sent, but their contents—except in special circumstances, the original variable is unchanged.

Lines 9-11: The turtle is moved back to the origin of the graph with the pen up and the heading reset to point straight up the screen.

Line 12: This loop draws a small line below the X axis and then calls DO_GRID to draw series of vertical lines with the turtle in Pen Erase mode. The effect is to create a grid of small gaps in the horizontal lines drawn so far.

Lines 13-17: Much the same procedure as lines 9-12 above. A grid is drawn with DO_GRID but this time the lines start from the Y axis instead of just to the left of it and the turtle is in XOR mode, which is short for ‘exclusive OR’. In this mode, any pixel which the turtle passes over is changed—if it was set before it is erased, if it was blank then it is set. What actually happens is that the horizontal lines are erased, except where the small gaps have been cut,

where a small dot will be placed, creating a regular grid of dots as shown on the illustration above.

Lines 18-24: The axes themselves are drawn.

4.3.3. Do__grid

The procedure which does the actual work of line drawing, based on information sent to it by GRID.

4.3.3.1. Listing

```
1:   to do_grid :len :width
2:   fd :len
3:   pu
4:   bk :len
5:   rt 90
6:   fd :width
7:   lt 90
8:   end
```

4.3.3.2. Commentary

Line 1: The variables sent are LEN and WIDTH, which will both be interpreted during the course of the procedure as distances to travel. It is not possible to say which is horizontal and which vertical since DO_GRID starts from the current turtle direction, whatever it is.

Lines 2-4: The turtle draws a line of LEN turtle units and then moves back to the start point with the pen up.

Lines 5-7: The turtle turns right by 90 degrees, moves WIDTH units and then turns left by 90 degrees, restoring the original direction.

4.3.4. Max

When determining how the graph will be placed on the screen, the program needs to know what the biggest item of data to be placed on the Y axis is—only then can it know how much to scale down the figures to fit them onto the screen. It would clearly not make any sense to give the same scale to a graph where the data was in thousands and one where it was in tens. This little procedure extracts the largest item from the data supplied and can be used to do the same in any program.

4.3.4.1. Listing

```
1:  to max :data
2:    (local "big)
3:    make "big first :data
4:    make "data bf :data
5:    repeat count :data [if (item 1 :data > :big)
      [make "big item 1 :data] make "data bf :data]
6:    op :big
7:    end
```

4.3.4.2. Commentary

Line 1: The only variable supplied to the procedure is DATA, which is a list containing all the data for the graph.

Lines 3-4: The variable BIG is set to equal the first item of data in the list and the first item sliced off using the BF (BUT FIRST command). Note that this does not destroy the list of data, only the copy received when it was sent to the procedure.

Line 5: This loop, which is repeated once for every item of data in the list, carries out the following actions:

>> The loop is repeated for as many times as there are items in the list of data.

>> The first item in the list is compared with BIG and if it is larger, BIG is set equal to the larger value.

>> The first item is sliced off the list and the loop repeated.

Line 6: The final value of BIG, which must be the largest item in the list, is returned to the calling procedure. When you come to enter the next procedure you will notice that MAX is not simply used as a keyword, it included in a simple mathematical expression as if it were a variable. The value that it will take in the expression is whatever is returned by this line.

4.3.5. Dograph

This is the procedure which actually draws the line graph on the screen. The main part of the work is carried out by a single line loop.

4.3.5.1. Listing

```
1:   to dograph :size :data
2:   (local "x" width "yscale)
3:   make "yscale (item 2 :size) / max :data
4:   make "x -1
5:   pu
6:   setpos :pos
7:   pd
8:   make "width (item 1 :size) / ((count :data) -
   1)
9:   repeat (count :data) [make "x :x + 1 setpos (
   list :x * :width + (item 1 :pos) (item :x +
   1 :data) * :yscale + (item 2 :pos))]
10:  end
```

4.3.5.2. Commentary

Line 1: The variables received by the procedure represent the values for the size of the overall graph and the list containing the data.

Lines 3-8: These lines calculate the scale factors by which each item of data will be multiplied in order to ensure that the largest data items will fit onto the Y axis and the number of data items does not spill over the X axis. They also position the pen at the origin of the graph.

Line 9: This loop carries out the following actions:

>> The loop is repeated for as many times as there are items of data in the list DATA.

>> The variable X, which records the number of items processed so far, is incremented by one—it is set outside the loop to minus one.

>> The turtle is re-positioned to a horizontal position of X times the width of divisions along the horizontal axis plus the position of the origin of the graph. Vertically, the position will be the value of item X + 1 in the list of data, multiplied by the scale of the units on the vertical axis plus the position of the origin of the graph. Since the pen is down while all this is being done, simply re-positioning the turtle extends the line making up the graph

>> The loop is repeated until no more items of data remain to be charted.

4.3.6. Countchar

One of the elementary features which is missing from most versions of Logo is the capacity to count the number of characters in a string—because there no such things as strings, only lists. This procedure is here to count the number of characters in the title that the graph is given, so that it can be properly centred on the screen.

4.3.6.1. Listing

```

1:  to countchar :data :c
2:    (local "i)
3:    if (wordp :data) [op (count :data) + :c]
4:    make "i 1
5:    repeat count :data [make "c countchar item :
      i :data :c make "i :i + 1]
6:    op :c
7:    end

```

4.3.6.2. Commentary

Line 1: The variables supplied are:

DATA: The name of the data to be examined—in our case we shall be sending a word or words.

C: The number of characters so far counted—this makes no sense the first time COUNTCHAR is called, but eventually it will call itself and tell itself how many characters have been counted so far—why will be explained in the commentary following.

Line 3: If there is only one word, ie, no spaces, in the text being examined, then this will be indicated by the WORDP function and COUNT is used to return to the calling procedure the number of items of data, in this case letters, in the word.

Line 5: The reason for this loop is that if the text being examined contains spaces, Logo regards it as a series of different lists and COUNT cannot be used to obtain a fast overall total of characters. In that case COUNT is used to obtain the number of lists (words separated by spaces) and then each word is sent to COUNTCHAR to have the individual characters counted.

The procedure is quite indifferent to the fact that it is being called up by itself, a process called recursion, and will happily count the characters of the individual words it is sent by itself. No confusion, except perhaps in the minds of programmers, arises out of all this because of the way that the variables a procedure uses are special to it and do not alter variables, even those of the same name, used elsewhere. When COUNTCHAR Mark I calls up COUNTCHAR Mark II, a completely different set of variables is used and the only result is that COUNTCHAR Mark I gets back a number of values for the length of words which it totals into the length of the title.

4.3.7. Htext

This procedure handles the placing of the title on the screen, properly centred, in the text area at the bottom.

4.3.7.1. Listing

```
1:   to htext :text :y
2:   setcursor (list (45 - ((countchar :text 0) /
   2)) :y)
3:   type :text
4:   end
```

4.3.7.2. Commentary

Line 1: The variables supplied are the title itself and the number of lines down the screen where it is to be placed.

Line 2: The cursor is positioned so that it is half the length of the title to the left of the mid point on the 90 character screen—ie, when the title is printed it will appear centred on the screen.

4.3.8. Line__test

After everything that you have already entered, finally comes the essential procedure which calls up the main part of the program and supplies it with the data to draw the graph. In use, you would edit this procedure, or a copy of it called something like L__GRAPH and then call it from the command line in order to draw the graph itself. You can of course have several different modules with different names, all of which can contain different data which they send to the main part of the program. Provided that each procedure calls

up the LINEGRAPH procedure, supplies it with the correct data and that all the procedures in this program have been defined by loading them into memory, you can draw a variety of different graphs in succession.

4.3.8.1. *Listing*

```

1:   to line_test
2:   cs
3:   pu
4:   setpos [-200 -75]
5:   linegraph [Test graph] [400 220 10] [5 10 20
    40 70 110 140 160 170 175 170 160 140 110 70
    40 20 10 5]
6:   end

```

Line 4: The origin at which you wish your graph to start—the rest of the program will accept this point as given, so it only has to be set once here.

Line 5: The three sets of material in square brackets represent:

- 1) The graph title.
- 2) The lengths, in turtle units, of the X and Y axes and the distance between each marker on the Y axis—the spacing on the X axis will depend upon the number of items.
- 3) The data for the graph.

All of these items you can edit for yourself to create your own graph, as long as you supply one item of text in the first set of brackets, three figures in the second, and any number of data items in the third. Not all values will produce a sensible graph, of course, but the program will at least try.

4.3.8.2. *Testing*

Having saved the program, enter LINE__TEST from the command line and the program should create the line graph illustrated at the beginning of this section. You are then free to try defining some other start-up procedures based on LINE__TEST, since the program is now ready for use.

4.4. B__Graph

The second program in this section is a good example of the way in which material which is properly written in clearly laid out procedures is easily adaptable to other purposes. The illustration below indicates that the purpose of the program is to create a histogram rather than a line graph but many of the procedures in this program are exactly the same as in the previous example. All that needs to be changed is the set of procedures which draw the line itself. If you wished to, the procedures to draw the bar graph could be tagged on to the end of the line graph program, so that after the line graph had been displayed and a key pressed, the specific bar graph procedures would come into play, calling up many of the line graph procedures for a second time to create the new display.

Since so much of the program is identical to what has gone before, we shall not waste space by listing the whole program. What you will find given below is a list of the modules which are different from those in the previous program, together with the names of the procedures which they are intended to replace. To make up the B__GRAPH program, delete from the L__GRAPH program the following procedures:

LINEGRAPH
DO__GRAPH
LINE__TEST

Having done that, enter the procedures shown in this section and save the whole program under a new name.

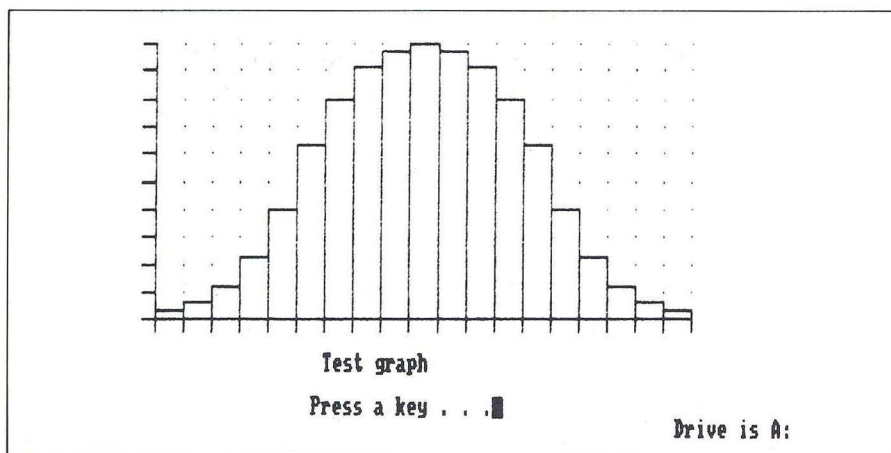


Fig. 4-2: A typical B__GRAPH display

4.4.1. Bargraph

This module is identical to the LINEGRAPH procedure in the previous program, except that the DO_BAR procedure is called rather than DOGRAPH.

4.4.1.1. Listing

```

1:   to bargraph :title :size :data
2:     (local "pos "loop "t)
3:     setsplit 3
4:     ss
5:     ct
6:     make "pos (list first tf item 2 tf)
7:     ht
8:     pu
9:     htext :title 28
10:    grid first :size item 2 :size count :data
    item 3 :size
11:    dobar :size :data
12:    htext [Press a key . . .] 30
13:    label "loop
14:    if keyp [make "t rc] [go "loop]
15:    setsplit 10
16:    ct
17:    end

```

4.4.2. Dobar

The only really different module in the whole program, this one draws the boxes which make up the histogram.

4.4.2.1. Listing

```

1:   to dobar :size :data
2:     (local "high "width "yscale)
3:     make "yscale (item 2 :size) / max :data
4:     pu
5:     setpos :pos
6:     pd
7:     make "width (item 1 :size) / (count :data)
8:     seth 0
9:     repeat (count :data) [make "high (first :

```



```
data) * :yscale fd :high rt 90 fd :width lt
90 bk :high make "data bf :data]
10: end
```

4.4.2.2. *Commentary*

Lines 1: The variables sent to the procedure consist of:

SIZE: The list of three items representing the length of the two axes and the number of divisions on the Y axis.

DATA: The list of data on which the graph is to be based.

Line 3: The scaling of the Y axis is determined according to the length of the axis and the maximum value to be expressed.

Line 7: The width of the divisions on the X axis is determined by the length of the axis and the number of items of data,

Line 9: This one line draws all the boxes which make up the histogram. The process is as follows:

>> The loop is repeated for as many times as there are items of data.

>> The variable **HIGH** is set to the value of the first item of data multiplied by the scaling factor **YSCALE**.

>> The turtle draws a line **HIGH** units long, turns to the right and draws the top of the box **WIDTH** units wide, turns to the left and moves back down to the X axis, completing the box.

>> Finally the list of data has the first item sliced off using the **BUT FIRST (BF)** command and the loop repeated.

4.4.3. **Bar__test**

The equivalent of the **LINE__TEST** module in the previous program except that this one calls up the **BAR__GRAPH** procedure. The data for the graph is identical.

4.4.3.1. Listing

```

1:   to bar_test
2:   cs
3:   pu
4:   setpos [-200 -75]
5:   bargraph [Test graph] [400 220 10] [5 10 20
    40 70 110 140 160 170 175 170 160 140 110 70
    40 20 10 5]
6:   end

```

4.4.3.2. Testing

With the program entered, type 'BAR__TEST' from the command line and you should see the histogram created. As with the previous program, different graphs can be created by editing the BAR__TEST procedure or by making copies of it under different names so that a variety of different graphs can be drawn, one after another.

If you want to be really clever, load the L__GRAPH program while you still have B__GRAPH in memory. You will now have the working procedures for both programs (some of them shared, of course) and be able to enter either 'BAR__TEST' or 'LINE__TEST' at the command line to draw either kind of graph.

4.5. Animals

A program in Basic or Logo called 'ANIMALS' probably exists on the computer systems of almost every university and college in the country. The reason for its spread is that it represents one of the first experiments into the field of artificial intelligence. Of course, the term artificial intelligence is a gross over-statement of what actually happens in most programs that claim the name, and the same is true of ANIMALS. Nevertheless the program is an interesting one because, in a limited way, it appears to the user to be able to learn and to use the knowledge that it gains in ways that are often unexpected.

Basically, the object of the ANIMALS program and its variants is to identify objects based on the answers to questions posed to the user. The name the program is usually given comes from the fact that its earliest versions were set the task of identifying an animal that the user had in mind, narrowing down the search by asking yes/no questions. While that particular task may not seem

to be of any great interest, the program has been valued around the world simply because it demonstrates computer learning. In addition, it is the prototype for the rapidly growing body of programs known as expert systems.

Such programs often use exactly the same techniques as ANIMALS to try and solve more important problems, up to and including medical diagnosis. In this version of the program, though we have affectionately kept to the traditional name, you will find that the specimen data included with the program actually applies the learning process to car maintenance problems, with the program attempting to pin-point problems which occur. You are free, without altering the program, to build up your own files of questions and answers to apply the program to a variety of different fields—the more time you are prepared to spend in devising sensible lists of questions and the facts that they point to, the better the program will perform.

4.5.1. List processing

Just as important as the interesting possibilities that the program opens up for you to build up your own simple expert system is the fact that it displays what is undoubtedly the best feature of Logo in serious use—list processing. In the programs given previously, we have made use of lists without any comment on their nature. While we cannot devote space to a complete analysis, a brief explanation will probably help in understanding the program which follows.

Probably the easiest way to underline the usefulness of lists is to compare them with the main data structure in Basic, the array. An array is a regular list of items which can have one dimension, in which case it stores one item after another as if in a line, two dimensions, in which case the data is stored as if in a table with height and width, or indeed many dimensions. Using multi-dimensional techniques it is possible to create quite very complex arrays. Dimensions do not have to be the same; you can declare an array such as `A(2,26,17)` as easily as any other.

Multi-dimensional arrays are a powerful tool in simplifying the storage and use of data. Suppose, for instance, you wished to compare the sales on 10 different companies on a weekly basis for a period of five years, in 7 seven different geographical areas. It sounds like a difficult problem to store the data in a way that will allow quick access but in fact the task is trivial. Simply declare an array such as `A(10,5,52,7)`. If you wish to know the performance of company three, in week 37 of year two, in area four, the figure is contained in element 3,2,37,4 of the array.

The problem with arrays is not that they are not powerful but that they are fixed. Very often in a program you do not know precisely what shape the data is going to take until it comes in. Even if you do know, it may not fit into the neat structure demanded by an array. Going back to the example of companies mentioned above, suppose there were only two companies and one worked in seven areas while the other only worked in three. The least you could get away with is an array of $2 \times 5 \times 52 \times 7$ elements. It would do the job for you but there is one major problem—a large amount of memory is wasted. For one of the companies there would be $4 \times 52 \times 5$ elements completely empty, representing the data for the four areas in which one company does not work. Totalling up, that is more than 1000 array elements simply sitting there using up some 5000 bytes of memory. For large and bodies of data the problem can literally become quite impossible and the data has to be broken down into more economical but less logical units, which require more programming before the data can be extracted.

The answer to complex data is lists. Simply defined, a list is a collection of items in numbered order. In Logo a list can be practically unlimited in length and the items can be numbers, characters or a mixture of both, without the system having to be told precisely how many items or of what nature they are before you begin. Not only that, items within a list can themselves be lists. What that means is that the data structure can be completely flexible, so that our two companies mentioned above could be covered by a list which goes as follows:

MAIN LIST: CONTAINING LIST 1 and LIST2

LIST1 and LIST 2: EACH WITH COMPANY NAME, AND LISTS FOR YEAR1, YEAR2 etc.

YEAR1, YEAR2 etc: EACH WITH LISTS FOR WEEK1, WEEK2 etc.

WEEK1, WEEK2 etc: EACH WITH FIGURES FOR AREA1, AREA2 etc.

Viewed diagrammatically, the data would look like this:

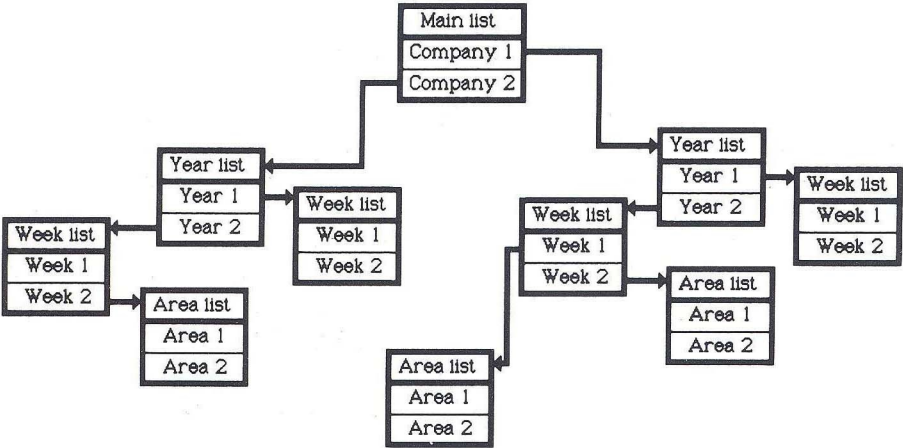


Fig. 4-3: Diagram of example data stored in lists

In this data structure, though it is not as clear cut as an array, no memory is wasted on unused elements since we only need to add sufficient elements to store the data we wish to hold.

4.6. Lists and binary trees

In the present program, lists are vital because we wish to use them to store a potential very wasteful structure known as a binary tree. A binary tree is a collection to questions and facts connected together by yes/no answers. Rather than explain in words, it is probably better to illustrate a binary tree with another diagram:

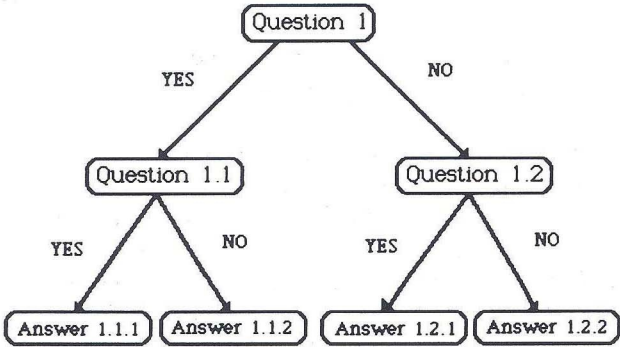


Fig. 4-4: Diagram of binary tree

The tree represents a body of data starting with a single fact or question (QUESTION 1). A yes or no decision, or any either/or decision is taken on the basis of the fact or question, and that leads to either QUESTION 1.1 or QUESTION 1.2. One the basis of those, further decision are made, leading to QUESTION 1.1.1, etc. Storing this data effectively in an array is a nightmare. Moreover, the whole point of the tree in the program that follows, and most others that use the structure, is that it can be added to. These additions will probably be irregular, with some of the tree's branches being very short and others growing very long as the program is used.

The whole task seems impossible until you see how simply the tree can be expressed in the form of a set of lists. Once again, the material in square brackets represents items which are themselves lists:

QUESTION1: TEXT OF QUESTION, [QUESTION 1.1], [QUESTION 1.2]

QUESTION 1.1: TEXT OF QUESTION, [QUESTION 1.1.1], [QUESTION 1.1.2]

QUESTION 1.2: TEXT OF QUESTION, [QUESTION 1.2.1], [QUESTION 1.2.2]

Suddenly, the whole thing becomes more manageable, both in terms of common sense and, perhaps more importantly, in terms of programming. Of course, the ability to store items in lists is not the end of the matter. Logo provides a range of facilities to allow the programmer to access the contents of lists in convenient ways, but it is the lists themselves which provide the real power, as the ANIMALS program demonstrates well.

In the case of this program, each list will contain the text of a question which the program will ask the user, together with two other lists which will either contain further questions or a guess as to the solution to the problem that has been set. The classic example of the program in action is suggested by the name ANIMALS, since in its traditional form it will ask the user a series of questions like 'Does it have four legs', 'is it carnivorous' and so on, until it comes to a point in the binary tree where instead of another question there is a suggested answer, such as 'dog'. If the suggested answer is correct, nothing more is done but if it is wrong then the program will ask the user for a new yes/no question which will distinguish between a dog and the new animal that shares the qualities of a dog but isn't one, together with the name of the animal. The new question is added to the tree, with one answer pointing to 'dog' and another pointing to the new animal.

Quite apart from being a useful way of looking at the capabilities of Logo, the program is also both enjoyable and potentially useful. Children especially love playing with it, especially as it builds up a body of knowledge and begins to guess more accurately the animal they are thinking of. In practical terms it can be used wherever a problem can be solved by the answers to a series of yes/no questions. Fault tracing of all kinds often comes under this heading, as does choosing between a wide range of products on the basis of differences of specification between them. The uses of the program, or something like it, are really only limited by your own ingenuity.

4.6.1. Getname

When using the program, you will not want to enter a set of questions and answers afresh every time so, as with the data handling programs in Basic, the program is capable of saving a set of questions held in memory and loading them again at a later time. This module accepts a file name or prints out the current disc directory if '?' is input.

4.6.1.1. Listing

```
1:   to getname
2:     (local "name)
3:     ct
4:     type [Enter filename (or ? for directory) :]
5:     make "name rq
6:     pr []
7:     if :name = "?" [(pr dir) pr [] type [Enter
      filename:] make "name rq pr []]
8:     pr []
9:     op :name
10:  end
```

4.6.2. Exist

This tiny procedure tests to see whether the file name input in the last module exists in the directory on the current disc.

4.6.2.1. Listing

```
1:   to exist :name
2:     op not empty? dir :name
3:     end
```

4.6.2.2. *Commentary*

Line 2: EMPTYP is a system function which returns a value indicating whether a list is empty or not. In this case we simply ask it if the directory of the disc, which is a list as far as Logo is concerned, contains the filename specified by the user and stored in NAME.

4.6.3. **Savefile**

The procedure to save a file of questions and answers—it will not overwrite an existing file of the same name, so if you wish to update a file you must alter the name before you re-save it.

4.6.3.1. *Listing*

```
1:   to savefile
2:     (local "name)
3:     make "name getname
4:     if not exist :name [recycle save :name] [
      quit [Name already exists - NOT saved]]
5:   end
```

4.6.4. **Loadfile**

The procedure to load a file back into memory.

4.6.4.1. *Listing*

```
1:   to loadfile
2:     (local "name)
3:     make "name getname
4:     if exist :name [load :name] [quit [File does
      not exist - NOT loaded]]
5:   end
```

4.6.5. **Quit**

This procedure is used when the load or save file procedures terminate with an error—it beeps at the user and prints a message supplied by the calling procedure.

4.6.5.1. Listing

```
1:  to quit :text
2:  pr (list char 7)
3:  pr :text
4:  pr []
5:  wait 23
6:  end
```

4.6.5.2. Commentary

Line 2: Printing character number seven produces a short beep.

4.6.6. Cleardata

If during the course of using the program, the user decides to start a fresh set of questions, the existing set must be cleared from the memory, a task accomplished by this module.

4.6.6.1. Listing

```
1:  to cleardata
2:  ct
3:  pr [Warning this function clears all current
    data from memory]
4:  pr (list char 7)
5:  type [Do you wish to continue (y or n) ?]
6:  if yes [make "data (list [] [] [])]
7:  pr []
8:  end
```

4.6.6.2. Commentary

Lines 6: If, when the user has cleared the memory, the session is going to continue with the entry of fresh questions, the basic list on which the eventual binary tree will be built needs to be defined. It consists of three items—we do not need to say at this stage what they are.

4.6.7. Question

Before the program gets down to asking the series of questions which will

hopefully result in it discovering the correct answer to the problem you have set it, these two procedure check that the lists have been validly set up and that there is at least one question to set the process rolling. If either is not true then the problem is dealt with first.

4.6.7.1. *Listing*

```

1:  to question
2:  ct
3:  if (empty? plist "data) [make "data [] [] []
   ]
4:  if (empty? plist "problem) [make "problem [
   problem]]
5:  if empty? item 1 :data [make "data first_
   answer :data] [make "data do_question :data]
6:  end

```

4.6.7.2. *Commentary*

Line 3: The empty list called DATA is set up if it does not already exist.

Line 4: The variable PROBLEM is the name of the overall title the program gives to the questions it asks. The name is user defined and if it does not already exist, the procedure for defining it is called up.

Line 5: If the lists have been set up but there is no first question, the next procedure in this section is called to obtain one from the user.

4.6.8. **First__answer**

This module starts off the process of creating a tree by asking for the first problem.

4.6.8.1. *Listing*

```

1:  to first_answer :data
2:  pr se [Please give me a] :problem
3:  make "data (list r1 [] [])
4:  op :data
5:  end

```

4.6.8.2. *Commentary*

Lines 2-3: If there is no first question, the program cannot do anything at all of any use. These lines accept characters from the keyboard and make them into the first item of the basic list called DATA.

4.6.9. **Do__question**

This short procedure is the core of the whole program, since these few lines are all that is needed to move down the largest binary tree, asking questions until a suggested answer is found. The procedure is a recursive one, meaning that it calls itself up in order to carry out further processing on the data. We looked at a very limited form of recursion in the CHARCOUNT module in the last two programs, but in this case the recursion is far more extensive, with the procedure calling itself up several times over. The very shortness of the procedure illustrates both the power of binary trees and how well they can be stored in the form of lists.

4.6.9.1. *Listing*

```
1:   to do_question :data
2:   (local "exit)
3:   if is_ans :data [make "data do_ans :data go "
   exit]
4:   pr []
5:   type se (item 1 :data) [(y or n) ?]
6:   if yes [make "data (list first :data do_
   question item 2 :data item 3 :data)] [make "
   data (list first :data item 2 :data do_
   question item 3 :data)]
7:   label "exit
8:   op :data
9:   end
```

4.6.9.2. *Commentary*

Line 1: The single variable sent this procedure is a list of the kind described in the introduction to the program, containing three items, the first of which will normally be a question while the second and third items are further lists of the same kind.

Line 3: Sometimes, the list supplied will not have a question and two lists, it will simply have one item, a suggested answer. In that case this line calls up the module which suggests answers and then exits the current procedure.

Line 5: Normally the first item on the list will be a question, and this is displayed on the screen.

Line 6: This is the line which carries out the recursion. The steps are as follows:

>> The procedure has received a list called DATA, which consists of a first item which is a question and two further items which are themselves lists representing what follows down the tree from 'yes' and 'no' answers.

>> If the later procedure YES indicates that the answer to the question is 'yes' then the list called DATA is altered so that it now consists of the list following on from a 'yes' answer. This new list itself consists of a further question and its 'yes' and 'no' lists.

>> If the answer is 'no', DATA is altered so that it now consists of the list following on from a 'no' answer.

>> The amended list is now sent to DO_QUESTION again so that the next question can be asked.

>> This process of calling itself up may go on many times before the end of the chain is found in the form of an answer. When that happens, the DO_QUESTION which came to the end of the chain hands back its list (which will have been changed if the answer was wrong) to the DO_QUESTION which called it, which hands it back up....and so on until the very first DO_QUESTION receives back the whole contents of the list.

If all of this is as clear to you as Ancient Egyptian poetry, do not fear. It is a painful fact that while it is very easy to describe the theory of recursion it is often very, very difficult to cast the mind around exactly what a recursive procedure is doing. The following example, where all the lists are spelled out in full, may help. The example is based on the binary tree illustrated:

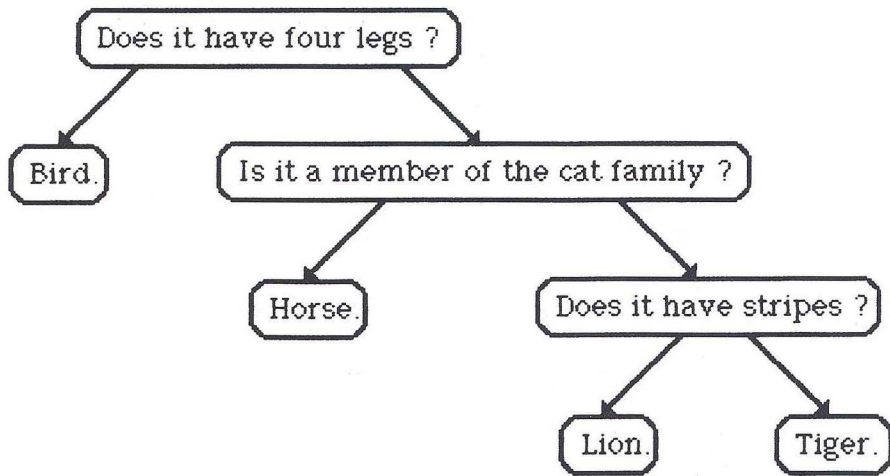


Fig. 4-5: Binary tree to identify a dog

The steps for dealing with the problem are as follows:

>> DO_QUESTION receives the list DATA, consisting of: 'DOES IT HAVE FOUR LEGS', [YES LIST] and [NO LIST].

>> The question is asked and the answer is 'yes'.

>> DO_QUESTION calls itself up and sends itself YES LIST.

>> DO_QUESTION (2) receives the list and calls it DATA, it consists of: 'IS IT A MEMBER OF THE CAT FAMILY' [YES LIST] [NO LIST].

>> The question is asked and the answer is 'no'.

>> DO_QUESTION calls itself up and sends itself NO LIST.

>> DO_QUESTION (3) receives the list and calls it DATA, it consists of: 'HORSE' [] []

>> Since there is nothing in the two lists, the program knows that this is meant to be an answer, so line 3 uses later procedures to suggest this. The answer comes back that the suggestion is wrong. In addition, the list received back from these later procedures has been changed so that it is no longer 'HORSE' [] [] but 'CAN YOU RIDE IT' [YES LIST] [NO LIST]. In fact the YES LIST is the answer 'horse' and the NO LIST is the answer 'dog'—the

changes have been made by the later procedures but DO_QUESTION does not need to know anything about them.

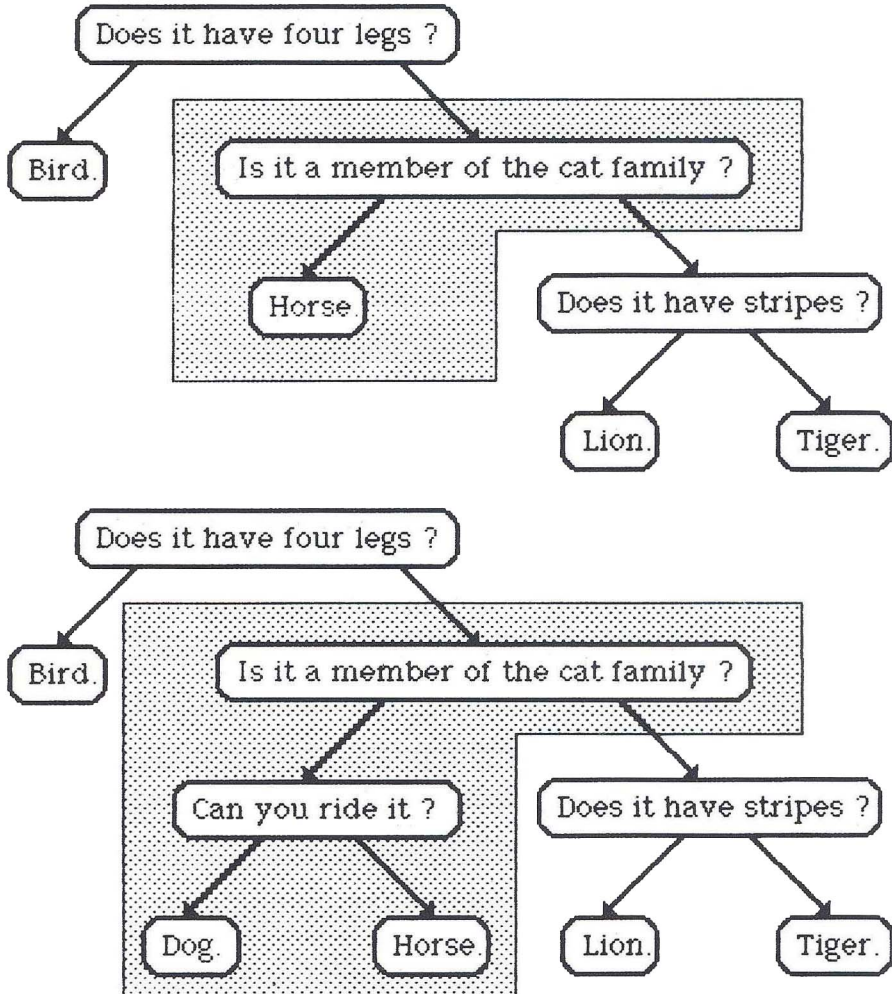


Fig. 4-6: The section of the tree before and after updating

>> DO_QUESTION (3) hands back its amended DATA list to DO_QUESTION (2)

>> DO_QUESTION (2) receives back a list from DO_QUESTION (3) but regards it as the current edition of NO LIST, which is what it sent to (3).

>> DO_QUESTION (2) hands back its DATA list, including the amended NO LIST to DO_QUESTION (1).

>> DO_QUESTION (1) receives back a list from DO_QUESTION (2) but regards it as the current edition of the YES LIST, which is what it sent to (2).

>> DO_QUESTION (1) hands back the completed DATA list to the rest of the program.

The result of all this is far simpler than the explanation. Either the program has suggested the correct answer to the user or the binary tree of questions and answers has been updated to include a new question and the appropriate answer.

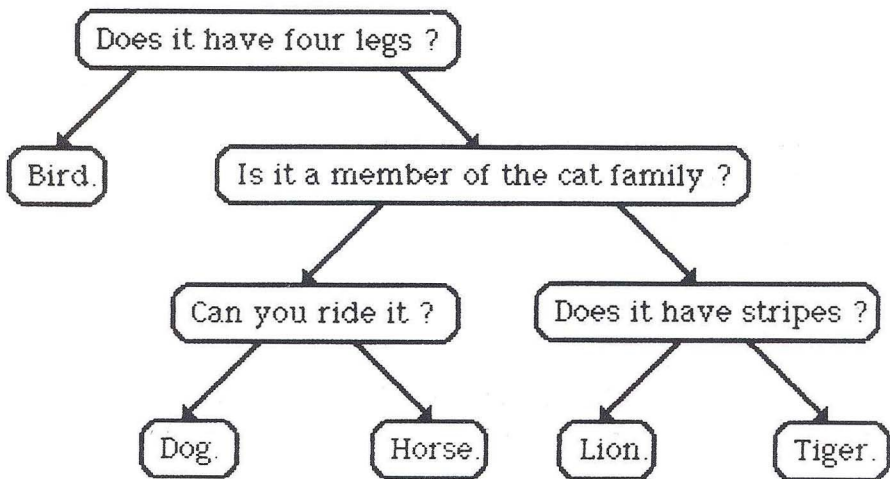


Fig. 4-7: The final binary tree

4.6.10. Is__ans

This procedure looks at a list sent to it and determines whether it contains a question and two further lists proceeding from the 'yes' and 'no' answers, or merely an answer and two empty lists. The output of the procedure is used by DO_QUESTION in determining whether it has reached the end a chain of questions.

4.6.10.1. Listing

```
1:   to is_ans :data
2:   op and (empty item 2 :data) (empty item 3 :
      data)
3:   end
```

4.6.11. Yes

A simple procedure which asks the user to input either 'Y' or 'N' and passes the result back to the calling procedure.

4.6.11.1. Listing

```
1:   to yes
2:   (local "loop "key)
3:   label "loop
4:   make "key rc
5:   if or (:key = "y) (:key = "Y) [type char 32
      pr [Y] op "TRUE]
6:   if or (:key = "n) (:key = "N) [type char 32
      pr [N] op "FALSE]
7:   go "loop
8:   end
```

4.6.12. Do__ans

When the DO_QUESTION procedure arrives at an answer rather than a question, it calls upon this procedure to suggest the answer to the user.

4.6.12.1. Listing

```
1:   to do_ans :data
2:   pr []
3:   pr se (se [I think the] :problem) (se [is]
      item 1 :data)
4:   pr []
5:   type [Is this correct (y or n) ?]
6:   if not yes [make "data add_ans :data]
7:   pr []
8:   op :data
9:   end
```


4.6.12.2. *Commentary*

Line 6: If the answer suggested to the user is rejected, the subsequent ADD__ANS procedure is called to add a new answer to the tree.

4.6.13. Add__ans

This is the procedure that adds a new answer to the tree when an answer which appears to fit the questions asked is rejected by the user. The procedure is not in any way complex, but the number of questions which have to be asked make it longer than average.

4.6.13.1. *Listing*

```
1:   to add__ans :data
2:   (local "y__ans "n__ans "loop "temp)
3:   label "loop
4:   pr []
5:   make "n__ans item 1 :data
6:   type [What is the correct answer ?]
7:   make "y__ans r1
8:   pr []
9:   type [What question will distinguish between]
10:  type (list char 32)
11:  type :n__ans
12:  type (list char 32)
13:  type [and]
14:  type (list char 32)
15:  type :y__ans
16:  pr [?]
17:  make "quest r1
18:  pr []
19:  type [If you reply Yes to]
20:  type (list char 32)
21:  pr :quest
22:  type [Should I use answer]
23:  type (list char 32)
24:  type :y__ans
25:  type (list char 32)
26:  type [(y or n) ?]
27:  if not yes [make "temp :y__ans make "y__ans :n
               ans make "n__ans :temp]
```

```

28:   pr []
29:   type [Is this correct (y or n) ?]
30:   if not yes [go "loop]
31:   pr []
32:   make "y_ans (list :y_ans [] [])
33:   make "n_ans (list :n_ans [] [])
34:   op (list :quest :y_ans :n_ans)
35:   end

```

Lines 6-7: At this point the program has asked the user a question like 'Is it a dog' and received the answer 'no'. The first thing that needs to be done is request the user to input the correct answer arising out of the series of questions which have been asked.

Lines 9-17: The problem that now has to be dealt with is that on the basis of its current knowledge the program cannot distinguish between the current answer, eg 'dog' and the new answer just supplied by the user. Both of them are at the end of exactly the same chain of yes/no answers. The user is therefore asked supply another yes/no question which, when answered, will distinguish between a dog and the new answer.

Lines 19-30: The user has now supplied a new question like: 'Does it have hooves'. The program has no way of knowing whether a 'yes' answer to this question would point to a dog or to the new answer. These lines ask the user which of the two a 'yes' answer would indicate and then place the answer pointed to by 'yes' into the variable Y_ANS and the other into the variable N_ANS.

Lines 32-34: The final act of the procedure is to add the new question and its two answers to the binary tree. This is done by turning the two answers into lists of the correct kind and then adding them to the list of which the first item is the new question. The DO_QUESTION procedure will then receive the new list back into the tree. The process is illustrated in the diagram at Fig. 4.6.

4.6.14. Setprompt

The name which the program applies to the answers it supplies is selected by the user. In the case of questions about animals, the prompt would probably be 'Animal', in other cases it might be 'Problem' or 'Suitable Component'. This simple module allows the user to input the chosen prompt.

4.6.14.1. Listing

```
1:  to setprompt
2:  (local "temp)
3:  ct
4:  pr se [The current prompt is] :problem
5:  pr []
6:  type [Enter new prompt for question ('!'
       keeps current prompt) ?]
7:  make "temp r1
8:  if (not (:temp = [])) [make "problem :temp]
9:  end
```

4.6.15. Menu

You have already seen a great many menus in the course of the chapter on Basic. This is an equivalent module in Logo.

Information System

Commands available:

- 1) Ask questions
- 2) Save current data
- 3) Load new data
- 4) Reset question prompt
- 5) Initialise data
- 6) Quit program

Enter Command required:■

Drive is A:

Fig. 4-8: The ANIMALS menu

4.6.15.1. Listing

```
1:   to menu
2:     (local "command)
3:     ts
4:     label "main
5:     ct
6:     setcursor [30 1]
7:     pr [Information System]
8:     setcursor [25 5]
9:     pr [Commands available:]
10:    setcursor [30 9]
11:    pr [1) Ask questions]
12:    setcursor [30 11]
13:    pr [2) Save current data]
14:    setcursor [30 13]
15:    pr [3) Load new data]
16:    setcursor [30 15]
17:    pr [4) Reset question prompt]
18:    setcursor [30 17]
19:    pr [5) Initilise data]
20:    setcursor [30 19]
21:    pr [6) Quit program]
22:    setcursor [25 23]
23:    type [Enter Command required:]
24:    label "inputloop
25:    make "command (ascii rc) - 48
26:    if (or (:command < 1) (:command > 6)) [go "
inputloop]
27:    type :command
28:    if (not ((ascii rc) = 243)) [go "main]
29:    if (:command = 1) [question]
30:    if (:command = 2) [savefile]
31:    if (:command = 3) [loadfile]
32:    if (:command = 4) [setprompt]
33:    if (:command = 5) [cleardata]
34:    if (not (:command = 6)) [go "main]
35:    cs
36:    end
```


4.6.15.2. *Commentary*

Line 28: It is worth noting that due to the way that Logo redefines the keyboard, the Return key in Logo has the value 243 in the character set, rather than the 13 which is usual in Basic and most other situations.

4.6.16. Some specimen data

If you save the program without clearing the data using menu option five, Logo will automatically add all the data to the end of the program in the form of a series of MAKE statements. We have left the lines in the listing to give you a small set of questions in the area of car mechanics to experiment with. In normal use, you will choose between saving the data in a separate file, thus allowing the program to choose between data files, and saving the file together with the current set of data—both have their advantages.

4.6.16.1. *Listing*

```
1:  make "problem [car problem]
2:  make "data [[Does the starter motor turn] [[
    Does the petrol gauge show there is any
    petrol] [[Damp plugs] [] []] [[No petrol] []
    []]] [[Do the lights work] [[Starter motor
    broken] [] []] [[Flat battery] [] []]]]
```

4.6.16.2. *Testing*

You are now in a position to test your program by running it and supplying some problems relating to car mechanics in the first instance.

In developing the program for use it is worth bearing in mind that its analysis of a problem will only be as good as the questions you give it to work upon and, just as important, the *order* in which the questions are given. As you feed questions in, you should always be attempting to split the current problem in half. For instance, in an animals program, a question like 'Is it warm blooded' should always be first, since it splits the total stock of animals along a major division. A question such as 'Does it have six legs' may be very useful when you have come to the point where the animal is pinned down as some kind of insect but it doesn't divide the whole of the animal kingdom very neatly. The ideal ANIMALS program is one that looks at its own questions and re-arranges them in the order which gets down to solid answers quickly.

Index

A

ACCOUNTANT program, 45
accounts, 45
ANIMALS program, 225
arrays
 deleting items, 43
 inserting items, 36
artificial intelligence, 225
assembly language, 161
ASSIGN.SYS file, 157

B

BANKER program, 31
bar graph, 223
BASIC, 27
 auto-initialisation of programs, 34
 binary search, 64
 CALL command, 160
 control characters, 34
 cost of arrays in memory, 59
 cursor location, 34
 data files, 42
 degrees, 174
 error trapping, 90
 Jetsam commands, 28
 modular programming, 28
 program initialisation, 33
 program menu, 35
 radians, 174
 saving programs, 29, 32
 testing programs, 29
basic input/output system, 3
BASIC.COM, 13
binary search, 78

binary trees, 228

bios, 3

BLOCK program, 187

BUDGET program, 113

B_GRAPH program, 222

C

CARDINDX program, 85

CEN:, 8

CF2 discs, 7

command line interpreter, 4

CONOUT:, 8

CP/M, 3

 command line, 4

 DEVICE command, 8, 13

 DISCKIT program, 13

 ED program, 14

 editing keys, 24

 execution of GSX, 161

 file extensions, 24

 PIP command, 13

 structuring discs for, 9

 submit files, 14

 system attribute, 11

 table of common usage, 21

 USER command, 10, 17

 CP/M Plus, 6

 CP/M SETKEYS command, 209

 CPS8256 interface, 8, 16

 CRT:, 8

D

data

 loading from disc, 42

- saving to disc, 42
- data files, 42
- default disc drive, 7
- degrees, 174
- deleting items from arrays, 92
- DEVICE command, 8, 16
- DEVICE.COM, 8
- devices, 154
 - CEN:, 8
 - CONOUT:, 8
 - CRT:, 8
 - logical, 7, 154
 - LPT:, 8
 - LST:, 8
 - physical, 7
 - SIO:, 8
 - virtual, 154
- Digital Research Inc., 6, 183, 228
- disc drives, 7
 - change default, 22
 - default, 7
 - order of searching, 15
- DISCKIT, 13
- discs
 - copying, 22
 - formatting, 22
 - verifying disc copy, 23

E

- ED program, 14
- editing keys, 24
- Epson FX80, 157
- error trapping, 90
- expert systems, 225

F

- fields, 86
- files
 - copying, 22
 - copying between user groups, 22
 - copying sys files, 22
 - deletion, 22
 - display contents, 22

- print numbered text file, 23
- print text file, 22
- protecting, 22, 19
- read only status, 18, 19
- read/write status, 18, 19
- recommended extensions, 24
- renaming, 23
- unprotecting, 23
- filing, 64, 92

G

- GBASIC, 22
 - creation, 22, 163
- GENGRAF program, 163
- Graphics Extension System, 153
- GSX, 153
 - adjustment of height or width
 - of output, 179
 - arrays, 159
 - assembly language techniques, 161
 - ASSIGN.SYS file, 157
 - BLOCK program, 187
 - calling from BASIC, 160
 - capabilities on PCW, 199
 - clear workstation command, 166
 - close workstation
 - command, 165, 175
 - colour capability, 207
 - creating GBASIC, 163
 - device numbers, 157
 - draw polygon command, 168, 193
 - draw polyline command, 167, 193
 - draw polymarker command, 167
 - draw text command, 168, 180
 - error handling, 175
 - execution by CP/M, 161
 - fill patterns capabilities, 207
 - form of commands, 158
 - generalised drawing primitive
 - command, 168
 - GENGRAF program, 163
 - GKS Metafiles, 205
 - GSX__TEST program, 199

- information received back
 - from, 207
 - installation, 163
 - line drawing capabilities, 207
 - memory allocation, 158, 163
 - open workstation command, 165, 175
 - PIE program, 172
 - place graphics cursor command, 166
 - polymarker capabilities, 207
 - primitives capabilities, 207
 - remove last graphics cursor command, 167
 - set character height command, 108, 169
 - set fill index command, 171, 177
 - set fill style command, 171, 177
 - set polyline command, 169
 - set polymarker command, 170
 - set writing mode command, 171
 - table of commands, 164
 - text capabilities, 207
 - types of device, 207
 - update workstation command, 166, 176
 - GSX initialisation, 158
 - GSX workstations, 165
 - GSX_TEST program, 199
- H
- HELP, 22
- Hewlett Packard HP7470, 157
- I
- invoices, 74
- J
- Jetsam commands, 28
 - adding a record, 96
 - creating a file, 92
 - deleting a record, 111
 - deleting an index key, 110
 - index files, 99
- key rank, 100
- key set, 100
- opening an existing file, 95
- order of keys in index, 100
- searching a file, 102
- writing a record to a file, 99
- K
- keyboard
 - changing configuration, 23
- L
- line graph, 212
- list processing, 209, 226
- LocoScript, 1, 9, 20
- logical devices, 7
- logical disc drives, 5
- LOGO, 27, 209
 - binary trees, 228
 - ER command, 211
 - keyboard, 211
 - list processing, 209, 226
 - lists, 226
 - merging procedures, 225
 - passing of variables, 213
 - primitives, 210
 - procedures, 210, 211
 - recursion, 235
 - saving procedures, 211
 - saving programs, 211
 - setting up a disc, 210
 - turtle, 209
- LPT:, 8
- LST:, 8
- M
- Mallard BASIC, 46
- memory drive, 11
- menu, 35
- modular programming, 28
- mouse, 153
- multiple choice questions, 136

N

NNUMBER program, 74

O

operating system, 3

P

Papert

Seymour, 209

physical devices, 7

physical disc drives, 7

pie chart, 172

PIE program, 172

PIP command, 13

plotters, 157

printer, 8, 16

control mode, 26

device drivers, 157

screen dump, 26

selecting between screen and, 39

sending control sequence to, 23

PROFILE.SUB, 14, 16

R

radians, 174

ramdisc, 11

read only status, 19

read/write status, 19

records, 86

recursion, 235

reset, 11, 26

reverse video, 34

RPED editor, 17

S

screen

device driver, 154

freeze output to, 18

selecting between printer and, 39

screen dump, 26

screen size, 23

search path, 23

SET command, 16, 17, 18

SET24X80 command, 17

SETDEF command, 15

SETKEYS command, 16, 209

SETLST command, 17

SETSIO command, 16

SIO:, 8

start-up discs, 17

stock-taking, 74

submit files, 13, 14

running, 23

SUBMIT.COM, 20

system attribute, 11

system files, 14

T

three-dimensional graphs, 198

TRIVIA program, 136

turtle, 209

U

UNIFILE program, 58

USER command, 10, 17

user groups, 9

change, 23

copy between, 23

V

viewing directories, 23

virtual devices, 154

W

what-if techniques, 134

working disks, 17

Z

Z80 chip, 6

registers, 161

Amstrad's new computers, the PCW8256 and 8512, have introduced a new generation of users to the potential of the personal computer. With professional facilities at an unprecedented price. The PCW8256 and 8512 have appealed to those who insist on good value in the equipment they buy.

Building on that success, this book shows how the power of the PCW8256 and 8512 can be applied to far more than word processing. *The Amstrad Companion* is an authoritative guide to working with the CP/M operating system and to practical applications using the logo and basic programming languages. Included in the book are updated versions of programs which are being used in more than 20 countries around the world to file data, issue invoices, keep simple accounts, create budgets and display data in the form of graphs.

Based on long experience with machines costing a few hundred pounds to systems more than ten times the price of the PCW8256 and 8512, *The Amstrad Companion* represents a practical way forward for anyone who has realised the power of their machine and wants to put it to work in the most economical way possible.


The Authors

David Lawrence and Mark England have between them written more than a dozen computer books, and are Sunshine Books' most experienced authors. David Lawrence is the author of the best-selling *The Working Amstrad*.

GB £ NET +007.95

ISBN 0-946408-95-5

00795



9 780946 408955



SUNSHINE

£7.95 net

ISBN NO: 0 946408 95 5